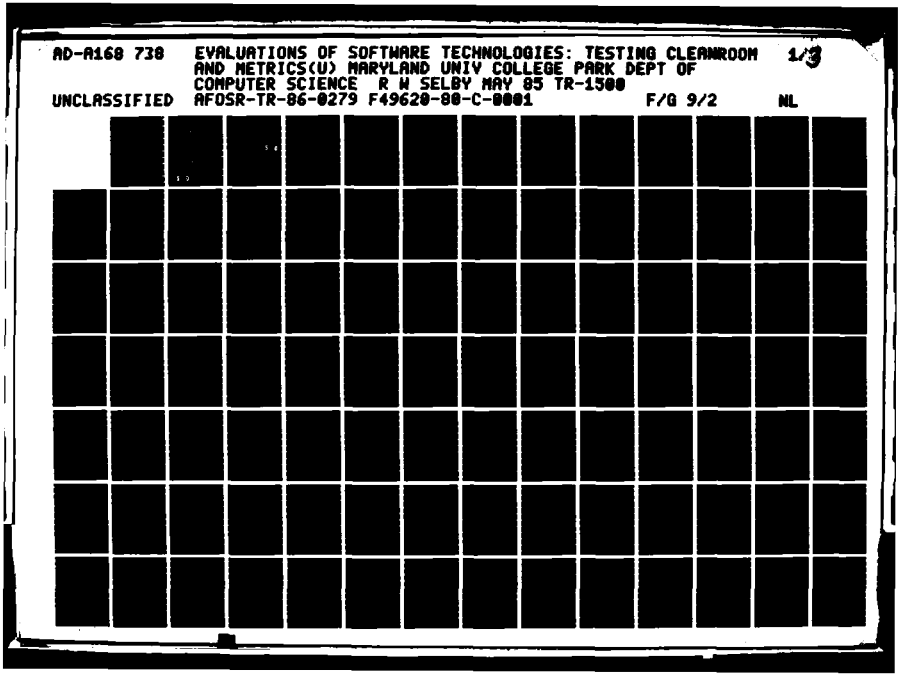
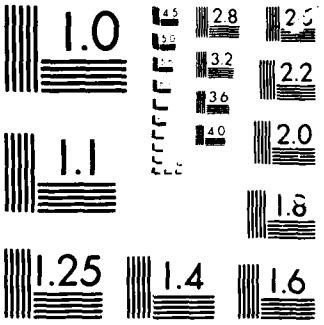
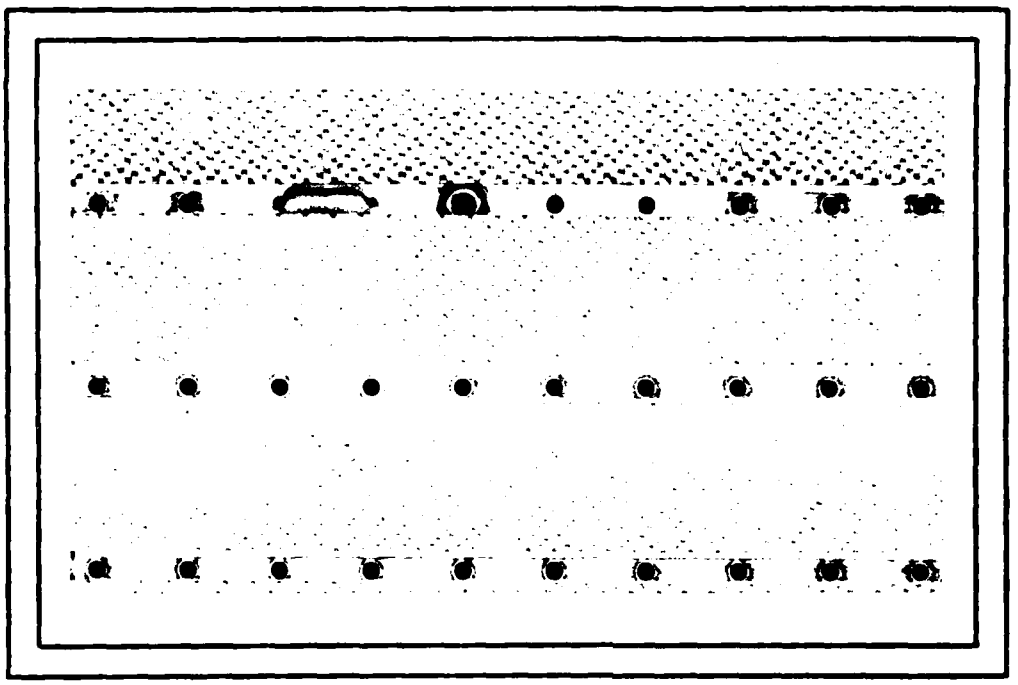

AD-A168 738 EVALUATIONS OF SOFTWARE TECHNOLOGIES: TESTING CLEANROOM AND METRICS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE R W SELBY MAY 85 TR-1500
1/3
UNCLASSIFIED AFOSR-TR-86-0279 F49620-80-C-0001 F/G 9/2 NL



AFOSR-TR- 86-0279

(2)



COMPUTER SCIENCE TECHNICAL REPORT SERIES

Approved for public release;
distribution unlimited.

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

Approved for public release;
distribution unlimited.

(2)

Technical Report-1500

May 1985

Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics

Richard W. Selby, Jr.

Department of Computer Science
University of Maryland
College Park

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Research supported in part by the AFOSR Contract AFOSR-F 49620-80-C-0001 and NASA Grant NSG-5123 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center and the Computer Science Center at the University of Maryland.

ABSTRACT

Title of Dissertation: Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics

Richard Wayne Selby, Jr., Doctor of Philosophy, 1985

Dissertation directed by: Victor R. Basili
Professor and Chairman
Department of Computer Science

The evaluation of software technologies suffers because of the lack of quantitative assessment of their effect on software development and modification. A seven-step approach for quantitatively evaluating software technologies couples software methodology evaluation with software measurement. The approach is applied in-depth in the following three areas. 1) Software Testing Strategies: A 74-subject study, including 32 professional programmers and 42 advanced university students, compared code reading, functional testing, and structural testing in a fractional factorial design. 2) Cleanroom Software Development: Fifteen three-person teams separately built a 1200-line message system to compare Cleanroom software development (in which software is developed completely off-line) with a more traditional approach. 3) Characteristic Software Metric Sets: In the NASA S.E.L. production environment, a study of 65 candidate product and process measures of 652 modules from six (51,000 - 112,000 line) projects yielded a characteristic set of software cost/quality metrics.

The major results are the following. 1) The approach described for quantitatively evaluating software technologies has been demonstrated and effective in

a variety of problem domains. 2) With the professionals, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 3) With the students, the three techniques were not different in the number of faults detected or in the fault detection rate, except that structural testing detected fewer faults than did the others in one study phase. 4) Code reading detected more interface faults and functional testing detected more control faults than did the other methods. 5) Most developers using the Cleanroom software development approach were able to build systems completely off-line. 6) The Cleanroom teams' products met system requirements more completely and succeeded on more operational test cases than did those developed with a traditional approach. 7) An approach described for calculating a characteristic metric set yielded the set for the NASA S.E.L. environment {source lines, design effort, number of input/output parameters, fault correction effort per executable statement, code effort, number of versions}.

ACKNOWLEDGEMENT

I greatly appreciate the opportunity to have worked with people that are shaping the frontier of the software engineering field. I wish to thank Larry S. Davis, John D. Gannon, Harlan D. Mills, and Kent L. Norman for serving on my committee and providing several insightful comments and suggestions. The ideas of code reading by stepwise abstraction and Cleanroom software development are those of Harlan D. Mills, to whom I am grateful. I wish to thank F. Terry Baker for his essential role in the collection of a major portion of the data presented. The assistance provided by Frank E. McGarry and Jerry Page was crucial to the success of my analysis in the NASA Software Engineering Laboratory. I want to thank John D. Gannon for his refreshing attitude and valuable support. The members of our research group and several fellow graduate students have offered encouragement and helpful criticisms on this work. I also wish to thank Michael E. Fagan, David H. Hutchens, and Marvin V. Zelkowitz for several enlightening discussions. A special appreciation goes to my advisor, Victor R. Basili, whose motivation, leadership, and spirit made this work possible.

This research was supported in part by the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 and the National Aeronautics and Space Administration Grant NSG-5123 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center and the Computer Science Center at the University of Maryland.

Table of Contents

List of Figures

1. Introduction

Computer science is both a theoretical science and a practical science. A lot of work has been done studying theoretical aspects of computer science: determining optimum algorithms, formulating mathematical models, proving theorems, etc. However, little work has been done studying the practice of computer science – studying how the discipline of computer science is actually applied.

There are several motivations for studying the practice of computer science. Programs in practice are different than those in theory. The programs developed, maintained, and managed in practice tend to be large, unwieldy, and complex. Almost everyone associated with computer science has had an experience where he/she has said, "Wait a minute, that did not turn out the way that I thought it would!". Although there are insights into how the theory applies in practice, these insights have not always been correct. In the practice of computer science, few objects are viewed in isolation; there is a complex interaction among the programmer, methodology-tool-technique, and computer. For example, consider the area of software testing. The process of software testing has existed a long time. Testing is the most common way to attempt to show that a program does what it is intended to do. Several theoretical results has been published in the area of software testing. Yet, what is the best way to test a program – use a functional testing approach, a structural approach, a nonexecution-based reading process? The challenge is that the best approach is

not known. How is such a question answered?

The overall objective of this dissertation is to examine factors that contribute to software development and maintenance. The investigations undertaken adhere to two major themes. First, the factors studied should have a high potential benefit to the process of attaining aspects of software quality: requirement conformance, operational reliability, and modifiable source code. Second, the investigations should capture the effect of the factors precisely by characterizing and evaluating them quantitatively.

The three analyses presented are studies of 1) software testing, 2) Cleanroom software development (which will be described later), and 3) software metrics. The three studies are intended to advance the understanding of 1) the contribution of various software testing strategies to the software development process and to one another; 2) the relationship between introducing discipline into the development process (as in the Cleanroom approach) and several aspects of product quality (requirement conformance, operational reliability, and modifiable source code); and 3) the use of software metrics to characterize software environments and to predict project outcome.

The evaluation of software technologies has suffered because of the lack of quantitative assessment of their effect on software development and modification. This dissertation describes a seven-step analysis methodology that is intended to structure the process of evaluating software technologies. The analysis methodology provides a paradigm that is applicable in a variety of problem domains and is used in-depth in the three studies presented.

Section 2 describes an approach for quantitatively evaluating software technologies and classifies previous studies of software. Section 3 discusses the selection of the three investigations conducted. The problem formulation, data analysis, and results from the three studies are presented in Section 4. Section 5 summarizes the conclusions from this work.

2. A Quantitative Approach for Evaluating Software Technologies

Several techniques and ideas have been proposed to improve the software development process and the delivered product. There is little hard evidence, however, of which methods actually contribute to quality in software development and modification. As a consequence, many management decisions and research issues are resolved by inexact means and seasoned judgment, without the support of appropriate data and analysis. As the software field emerges, the need for understanding the important factors in software production continues to grow. The evaluation of software technologies suffers because of the lack of quantitative assessment of their effect on software development and modification.

This dissertation supports the philosophy of coupling methodology with measurement. That is, tieing the processes of software methodology use and evaluation together with software measurement. The assessment of factors that affect software development and modification is then grounded in appropriate measurement, data analysis, and result interpretation. This section describes a quantitatively based approach to evaluating software technologies. The formulation of problem statements in terms of goal/question hierarchies is linked with measurable attributes and quantitative analysis methods. These frameworks of goals and questions are intended to outline the potential effect a technology has on aspects of software cost and quality. Problem formulation linked with the collection and analysis of appropriate data is pivotal to any management, con-

trol, or quality improvement process.

The analysis methodology described provides a framework for data collection, analysis, and quantitative evaluation of software technologies. The paradigm identifies the aspects of a well-run analysis and is intended to be applied in different types of problem analysis from a variety of problem domains. The methodology presented serves not only as a problem formulation and analysis paradigm, but also suggests a scheme to characterize analyses of software development and modification. The use of the paradigm highlights several problem areas of data collection and analysis in software research and management.

The approach described for quantitative evaluation of software technologies 1) applies a seven-step methodology for data collection and analysis, 2) couples problem formulation with quantitative analysis methods, and 3) suggests an analysis classification scheme. The following sections describe these aspects of the approach.

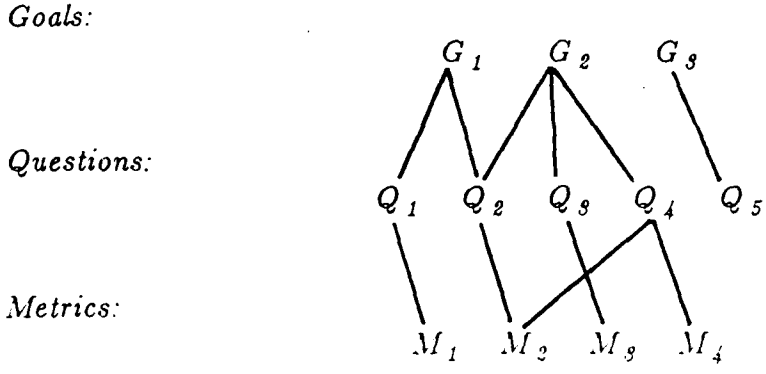
2.1. Methodology for Data Collection and Analysis

The methodology described for data collection and analysis has been quite useful. The methodology consists of seven steps that are listed below and discussed in detail in the following paragraphs (see also [Basili & Weiss 84]). 1) Formulate the goals of the data collection and analysis. 2) Develop a list of specific questions of interest. 3) Establish appropriate metrics and data categories. 4) Plan the layout of the investigation, experimental design, and statistical analysis. 5) Design and test the data collection scheme. 6) Perform

the investigation concurrently with data collection and validation. 7) Analyze and interpret the data in terms of the goal/question framework.

A first step in a management or research process is to define a set of goals. Each goal is then refined into a set of sub-goals that will contribute to reaching that goal. This refinement process continues until specific research questions and hypotheses have been formulated. Associated with each question are the data categories and particular metrics that will be needed in order to answer that question. The integration of these first three steps in a goal/question/metric hierarchy (see Figure 1) expresses the purpose of an analysis, defines the data that needs to be collected, and provides a context in which to interpret the data.

Figure 1. Goal/question/metric paradigm.



In order to address these research questions, investigators undertake several types of analyses. Through these analyses, they attempt to increase substantially their knowledge and understanding of the various aspects of the questions. The analysis process is then the basis for resolving the research questions and

for pursuing the various goals. Before actually collecting the data, the data analysis techniques to be used are planned. The appropriate analysis methods may require an alternate layout of the investigation or additional pieces of data to be collected. A well planned investigation facilitates the interpretation of the data and generally increases the usefulness of the results.

Once it is determined which data should be gathered, the investigators design and test the collection method. They determine the information that can be automatically monitored, and customize the data collection scheme to the particular environment. The several types of data that need to be collected usually require a data collection plan balanced across collection forms, automated measurement, and personnel interviews. After all the planning has occurred, the data collection is performed concurrently with the investigation and is accompanied by suitable data validity checks.

As soon as the data have been validated, the investigators do preliminary data analysis and screening using scatter plots and histograms. After fulfilling the proper assumptions, they apply the appropriate statistical and analytical methods. The statistical results are then organized and interpreted with respect to the goal/question framework. More information is gathered as the analysis process continues, with the goals being updated and the whole cycle progressing.

2.2. Coupling Goals With Analysis Methods

Several of the steps in the above data collection and analysis methodology interrelate with one another. The structure of the goals and questions should be

coupled with the methods proposed to analyze the data. The particular questions should be formulated to be easily supported by analysis techniques. In addition, questions should consider attributes that are measurable. Most analyses make some result statement (or set of statements) with a given precision about the effect of a factor over a certain domain of objects. Considering the form of analysis result statements will assist the formation of goals and questions for an investigation, and will make the statistical results more readily correspond to the goals and questions.

2.2.1. Forms of Result Statements

Consider a question in an investigation phrased as "For objects in the domain **D**, does factor **F** have effect **S**?". The corresponding result statement could be "Analysis **A** showed that for objects in the domain **D**, factor **F** had effect **S** with certainty **P**.". In particular, a question could read "For novice programmers doing unit testing, does functional testing uncover more faults than does structural testing?". An appropriate response from an analysis may then be "In a blocked subject-project study of novice programmers doing unit testing, functional testing uncovered more faults than did structural testing ($\alpha < .05$).".

Result statements on the effects of factors have varying strengths, but usually are either characteristic, evaluative, predictive, or directive. Characteristic statements are the weakest. They describe how the objects in the domain have changed as a result of the factor. E.g., "A blocked subject-project study of no-

vice programmers doing unit testing showed that using code reading detected and removed more logic faults than computation faults ($\alpha < .05$)." Evaluative statements associate the changes in the objects with a value, usually on some scale of goodness or improvement. E.g., "A blocked subject-project study of novice programmers doing unit testing showed that using code reading detected and removed more of the expensive faults to correct than did functional testing ($\alpha < .05$)." Predictive statements are a stronger statement type. They describe how objects in the domain *will* change if subjected to a factor. E.g., "A blocked subject-project study showed that for novice programmers doing unit testing, the use of code reading *will* detect and remove more logic faults than computation faults ($\alpha < .05$)." Directive statements are the strongest type. They foretell the value of the effect of applying a factor to objects in the domain. E.g., "A blocked subject-project study showed that for novice programmers doing unit testing, the use of code reading *will* detect and remove more of the expensive faults to correct than *will* functional testing ($\alpha < .05$)." The analysis process then consists of an investigative procedure to achieve the result statements of the desired strength and precision after considering the nature of the factors and domains involved.

Given any factor, researchers would like to make as strong a statement with as high a precision about its effect in as large a domain as possible. Unfortunately, as the statement applies to an increasingly large domain, the strength of the statement or the precision with which we can make it may decrease. In order for analyses to produce useful statements about factors in large domains,

the particular aspects of a factor and the domains of its application must be well understood and incorporated into the investigative scheme.

2.3. Analysis Classification Scheme

Two important sub-domains that should be considered in the analysis of factors in software development and modification are the individuals applying the technology and what they are applying it to. These two sub-domains will loosely be referred to as the "subjects," a collection of (possibly multi-person) teams engaged in separate development efforts, and the "projects," a collection of separate problems or pieces of software to which a technology is applied. A general classification of several software analyses in the literature can be obtained by examining the sizes of these two sub-domains that they consider.

Figure 2. Categorization of analyses of software.

# teams per project \ # projects	one	more than one
one	single project	multi-project variation
more than one	replicated project	blocked subject-project

Figure 2 presents this four part analysis categorization scheme. Blocked subject-project studies examine the effect of possibly several technologies as they are applied by a set of subjects on a set of projects. If appropriately configured, this type of study enables comparison within the groups of technologies, subjects, and projects. In replicated project studies, a set of subjects may separately apply a technology (or maybe a set of technologies) to the same project or problem. Analyses of this type allow for comparison within the groups of subjects and technologies (if more than one used). A multi-project variation study examines the effect of one technology (or maybe a set of technologies) as applied by the same subject across several projects. These analyses support the comparison within groups of projects and technologies (if more than one used). A single project analysis involves the examination of one subject applying a technology on a single project. The analysis must partition the aspects within the particular project, technology, or subject for comparison purposes.

Result statements of all four types mentioned above can be derived from all these analysis classes. However, the statements will need to be qualified by the domain from which they were obtained. Thus as the size of the sampled domain and the degree to which it represents other populations increase, the wider-reaching the conclusion.

The next section cites several software analyses from the literature and classifies them according to this scheme pictured in Figure 2.

2.4. Classification of Analyses of Software

Several investigators have published studies in the four general areas of blocked subject-project, replicated project, multi-project variation, or single project. The following sections cite analyses of the software development process and product from each of these categories. Note that surveys on experimental methodology in empirical studies have appeared in the literature [Brooks 80, Shell 81, Moher & Schneider 82].

2.4.1. Blocked Subject-Project Studies

[Curtis et al. 79] describe two experiments investigating factors that influence two aspects of software maintenance, understanding existing programs and accurately implementing modifications to them. The analyses involved the performance of 72 programmers operating on several versions of programs in three general software classes. The factors examined include control flow complexity, variable name mnemonicity, type of modification, degree of commenting, and the relation of programmer performance to various complexity metrics. They continued the investigation of how software characteristics relate to psychological complexity in [Curtis, Sheppard & Milliman 79]. This second paper describes a third experiment monitoring the ability of 54 programmers to detect different program bugs in distinct program versions.

[Hetzel 76] conducted a controlled experiment comparing different software testing techniques. The methods of functional testing, code reading, and a control group (both capabilities) were applied by 39 subjects to three different pro-

grams. In addition to describing technique performance, the testing strategies were related to factors of programmer background, self estimates of performance, and attitude.

[Miara et al. 83] describe a study to determine the contribution of indentation to program comprehensibility. The experimental approach examined the factors of level and type of indentation, as well as level of programmer experience. The understanding of seven different program variations was obtained by a comprehension quiz and a subjective rating of how difficult the program was to comprehend.

[Weissman 74] described several experiments conducted to measure a subject's ability to understand a program and his/her ability to modify it. The four areas of factors examined included aspects of program form, control flow, data flow, and interaction between control and data flow. The number of subjects studied ranged from 16 – 48. Each experiment used two different programs presented in varying combinations of the above factors. The measurements of understanding included self-evaluations, fill-in-the-blank quizzes, program hand simulation, ability to modify the program, and comprehension quizzes. The experiments were conducted sequentially to support the refinement of an appropriate experimental methodology.

[Gould & Drongowski 74] examined several factors related to computer program debugging: effect of debugging aids, effect of fault type, and effect of particular program debugged. Thirty experienced programmers separately debugged programs that contained a single fault. Three classes of faults in four

different one-page programs were used. Learning effects were examined and some possible "principles" of debugging were identified. Consistent results were obtained when the study was conducted on ten additional experienced programmers [Gould 75].

[Gannon & Horning 75] investigated the factor of language design and its relation to the reliability of the resulting software. Nine different language modifications were made to a programming language based on a analysis of its deficiencies. Two differently experienced subject groups completed implementations of two small but sophisticated programs (75-200 line) in the original language and in its modified version. The performance of the redesigned features in the two languages were contrasted in the frequency, type, and persistence of faults in the programs written by the subjects.

[Soloway & Ehrlich 84] examined two aspects of programming knowledge: programming plans and rules of programming discourse. Programming plans are generic program fragments that represent stereotypic action sequences in programming. Rules of programming discourse capture conventions in programming and govern the composition of the plans into programs. A total of 139 subjects participated in an experiment that required them to fill-in-the-blank in programs selected from four different software types. Some of the programs were written to violate certain hypothesized programming plans and discourse rules. A second similar study involving 41 professional programmers was conducted. The results in general support the existence and use of such plans and rules by both novice and advanced programmers.

Other blocked subject-project studies include [Panzl 81, Woodfield, Dunsmore & Shen 81].

2.4.2. Replicated Project Studies

[Basili & Reiter 81] present a study in which three different software development approaches are analyzed and compared. Seven three-person teams used a disciplined approach, six teams used an ad hoc approach, and six individuals used an ad hoc approach. Each of the 19 separate development efforts implemented a 1200 line compiler project. This allowed a comparison among the different development approaches, as well as among usability of various metrics for process measurement. A primary motivation for the experiment was to confirm certain beliefs of the beneficial effects of a particular disciplined methodology for software development. The researchers examined the factor of development technique and showed partial support for some of the beliefs by capturing several objective and automatable metrics of the development process and product.

[Johnson, Draper & Soloway 83] describe one of several studies done with the intention of characterizing misconceptions made by programmers and how they are manifested as bugs in programs. In this work they inspected the attempted implementations of an elementary problem by 204 novice programmers. They then classified the differences between the incorrect "buggy" programs and correct versions of similar structure. The differences were explained relative to mistakes in "programming plans" intended by the individual [Solo-

way et al. 82]. Further work comparing the factor of design strategies of novice and expert programmers is underway [Soloway 83].

[Bailey 84] presents preliminary observations from an experiment in which the Ada programming language was taught in two different fashions. [1] One class of subjects was first taught the high-level concepts supported in the language, such as modular design and data abstraction, and then taught the actual constructs and syntax of the language. The same material in a reverse order was presented to a second class of subjects; that is, constructs first and concepts second. Thus the factor studied was order of presentation of material. In addition to some preliminary exercises and academic scores, the two groups were compared on their ability to apply Ada in the design (only) of a small software system.

[Knight 84] examined the possibility of building ultra-reliable software systems by using N-version programming. The technique of N-version programming [Kelly 82] uses a high-level driver to connect several separately designed versions of the same system. The systems then "vote" on the correct solution, and the solution provided by the majority of the systems is output. The study examined 37 separately designed versions of the same 800 source line system. The factors examined included individual system reliability, total N-version system reliability, and classes of faults that occurred in systems simultaneously.

[Gannon 77] investigated the factor of static typing in programming languages. Two languages were generated which were essentially equivalent ex-

[1] Ada is a trademark of the U. S. Dept. of Defense.

cept for differences in the type conventions: one was statically typed (with integer and string types) and the other typeless (e.g., arbitrary subscripting of memory). A group of 38 subjects programmed the same problem in both languages, with half doing it in each order. The two languages were compared in the types of faults in the resulting programs, the number of runs containing faults, and the relation of subject experience to fault proneness.

[Shneiderman et al. 77] examined the factor of detailed flowcharts as an aid to program composition, comprehension, debugging, and modification. A series of five experiments was conducted on groups of 53 – 70 subjects of novice to intermediate expertise. A single program was used in all experiments except one, which used two programs. All experiments compared the performance in various programming tasks between groups that used some form of flowchart and those that did not. The performance of the groups was measured by comprehension quiz scores, correctness of programs written, correctness of modifications requested, and successful removal of seeded faults. No significant differences were reported between groups that used and those that did not use flowcharts.

[Parnas 72a] investigated the factor of proper system modularity as a means to eliminate the "integration phase" in development. A given system was decomposed into five modules, and four different types of implementation were specified for each module. Twenty subjects then independently developed the distinct implementations for the particular modules. At project completion, numerous combinations of the modules were assembled to form separate ver-

sions of the whole system. The minor effort required in assembling the systems evidenced support for the ideas on formal specifications and modularity discussed in [Parnas 72b, Parnas 72c].

[Boehm et al. 84] investigated the system development approachs of prototyping and specifying. Seven teams developed versions of the same application software system (2000 – 4000 line); four teams used a requirement/design specification approach and three teams used a prototyping approach. The final prototyped products were smaller, required less development effort, and were easier to use. The systems developed by specifications had more coherent designs, more complete functionality, and software that was easier to integrate.

[Myers 78] examined the factor of program testing technique and its relation to defect detection. The three techniques of 3-person walk-throughs, functional testing, and a control group were compared in the testing of a small (100 line) but nontrivial program. Fifty-nine data processing professionals were used as subjects. The techniques and their random pairings were compared in the number of faults found and their cost-effectiveness. The single techniques were not different in the number of faults they detected, while pairings of techniques were superior in terms of number of faults found.

Other replicated project studies include [Buck 81, Hwang 81, Hutchens & Basili 83].

2.4.3. Multi-Project Variation Studies

[Walston & Felix 77, Bailey & Basili 81, Basili & Freburger 81, Brooks 81, Basili, Selby & Phillips 83, Vosburgh et al. 84] are some of the numerous studies that have examined technological factors across several projects. The studies that consider separate development efforts coming from a single team or homogeneous environment genuinely belong in this category. Those that considered projects from a collection of heterogeneous teams or environments, such as [Vosburgh et al. 84], are placed here because they examined the differences in effect of the factors, but not the teams or environments. The factors investigated include structured programming, personnel background, development process and product constraints, project complexity, human and computer resource consumption, project duration, staff size, degree of management control, and productivity. In particular, [Bailey & Basili 81] mention 82 factors that could possibly affect project performance, including 36 from [Walston & Felix 77] and 16 from [Boehm 81]. They then describe a model generation process that uses a base-line of particular environmental aspects and captures differences among projects. The number of projects examined ranges from 18 in [Bailey & Basili 81] to 51 in [Walston & Felix 77, Brooks 81]. Among other results, these studies have led to increased project visibility, greater understanding of classes of factors sensitive to project performance, awareness of the need for project measurement, and efforts for standardization of definitions. Analysis has begun on incorporating project variation information into a management tool [Basili & Doerflinger 83].

[Bowen 84] examined the factors of estimating the number of residual faults in a system and of assessing the effectiveness of various testing stages. The study was based on fault data collected from three large (2000 – 6000 module) systems developed in the Hughes-Fullerton environment. The study partitioned the faults based on severity and analyzed the differences in estimates of remaining faults according to stage of testing.

[Adams 84] examined the factor of managing preventive service of software products in operational use. Preventive service constitutes installing fixes to faults that have yet to be discovered by particular users, but have been discovered by the vendor or other users. The study developed means to estimate whether and under what circumstances preventively fixing faults in operational software in the field was appropriate. The fault history for several large products (e.g., operating system releases, major components thereof) was empirically modeled.

[Vessey & Weber 83] examined the contribution of several factors to software maintenance: program complexity, programming style, programmer quality, and number of system releases. A total of 447 commercial and clerical Cobol programs in operation in one Australian organization and in two U.S. organizations were analyzed. The programs ranged from small to over 600 statements. In the Australian organization program complexity and programming style significantly affected the rate of maintenance repair. In the U.S. organizations the number of times a system was released significantly affected the maintenance repair rate.

2.4.4. Single Project Studies

[Endres 75, Basili & Weiss 81, Albin & Ferreol 82, Ostrand & Weyuker 83, Basili & Perricone 84] present the analysis of the distribution and relationships derived from change data collected during the development of a moderate to large software project. On a within project basis, they examined such factors as the frequency and distribution of faults during development, and their relationship with the factors of module size, software complexity, developer's experience, method of detection and isolation, phase of entrance into the system and observance, reuse of existing design and code, and role of the requirements document. Although conducted on only a single project, such analyses have produced fault categorization schemes and have been useful in understanding and improving a development environment.

[Gannon et al. 83, Basili et al. 85] examined a ground-support system written in Ada to characterize the use of Ada packages. Factors such as how package use affected the ease of system modification and how to measure module change resistance were identified, as well as how these observations related to aspects of the development and training.

[Basili & Ramsey 84, Ramsey 84] investigated the structural coverage of functionally generated input data. The functionally generated acceptance test cases and a sample of operational usage cases were analyzed from a medium-sized (10,000 line) satellite support system. The study examined the factors of the structural coverage of functional acceptance testing, the structural coverage

of operational product usage, the relationship between the program segments covered in acceptance testing and those covered in usage, and the relationship between structural coverage and fault detection.

[Baker 72a] analyzed the effect of applying chief programming teams and structured programming in system development. The large (83,000 line) system discussed is known as The New York Times Project. The project served as a field test for the new programming methodology concepts of structured code, top down design, chief programmer teams, and program libraries. Several benefits were identified, including reduced development time and cost, reduced time in system integration, and reduced fault detection in acceptance testing and field use.

3. Evaluation of Software Technologies: Problem Selection

The approach described in the previous section is intended to structure the process of analyzing software technologies by coupling software methodology evaluation with software measurement. The paradigm is applied in-depth in three different analyses. In addition to evaluating software technologies, the studies demonstrate the feasibility, utility, and effectiveness of the quantitative analysis paradigm. This section describes the selection of the different investigations conducted. The selection criteria for all the studies is discussed first, followed by an overview of each of the studies and how they apply the paradigm.

3.1. Selection Criteria

Three different studies were chosen to satisfy several criteria: scope of evaluation, domain sampling, quantitative analysis method, area of assessment, scope of technology, and potential benefit.

1) Scope of Evaluation – Each of the analyses should be a distinct type of study relative to the categorization of blocked subject-project, replicated project, multi-project variation, and single project. These different classes represent different pairings of the domain sizes of subjects and projects. Using the paradigm in these different categories shows its support for analysis of technologies across different scopes of evaluation.

2) Domain Sampling – The samples chosen from the subject and project domains in the studies should be representative of reasonably large populations.

Assessments using different sizes of software projects and different sizes of teams should be chosen. The selection and analysis of appropriate samples facilitates the extrapolation of the results to other environments, increases the usefulness of the results, and shows the performance of the paradigm in such situations.

3) Quantitative Analysis Method – Each of the studies should utilize a different method of quantitative analysis. Statistical techniques provide a soundly based, objective, and usually automatable mechanism to accomplish quantitative analysis tasks. Unfortunately, however, the amount of data required by some statistical approaches leaves them economically infeasible. Even with sufficient data, the generated results may yield unacceptable precision, too much unexplained variance, or doubt as to whether all the important factors are effectively captured. The use of this evaluation paradigm with a variety of quantitative methods demonstrates the flexibility of the approach across varying amounts and types of data.

4) Area of Assessment – The different problem areas investigated should not be precisely understood areas of software development and modification. The areas chosen should have open questions and unresolved issues. Selecting problems with these attributes provides a scenario similar to decision making situations in the field, where the proper outcome of the analysis is not known beforehand.

5) Scope of Technology – The analyses should examine technologies that are have distinct scopes of usage during software development and modification. Three different scopes of usage for technologies are a) individual technique: a

single technique used in conjunction with other techniques during a software project; b) development methodology: a system of methods that applies across the whole software project development; and c) environment methodology: a system of methods that applies across several projects in a development or modification environment. Using the paradigm in these categories demonstrates its effectiveness for evaluation of technologies having varying scopes of usage.

6) Potential Benefit – The analyses should address factors that can significantly contribute to the quality of the software development process and the developed product. The need for analysis of which factors contribute to quality in software development and modification is fundamental to the advancement of the field. The production of useful results from the use of the approach helps demonstrate its merit.

3.2. Analysis Selection

From the above set of criteria, three analyses were selected: 1) a comparison of software testing strategies; 2) an analysis of Cleanroom software development; and 3) a calculation of a characteristic software metric set. Figure 3 summarizes these studies relative to the criteria explained. Recall the use of the symbols from the previous section describing the quantitative methodology: D for domain sampled, F for factor or technology analyzed, and S for result statement type. As displayed in the figure, the above set of criteria are satisfied by the particular analyses selected. The following three sections discuss the application of the approach in the particular studies.

Figure 3. Three analyses selected.			
	Testing Study	Cleanroom Study	Characteristic Set
Scope of Evaluation	blocked subject-project	replicated project	multi-project variation
Subject Domains Sampled (D_1)			
Number	74 individuals	15 small teams (3-person)	1 medium environment (23-person)
Expertise	junior – advanced	junior – intermediate	junior – advanced
Project Domains Sampled (D_2)			
Number	4	1	6
Size	unit (160 – 350 LOC)	small system (1200 LOC)	large system (51,000 – 112,000 LOC)
Quantitative Analysis Method	fractional factorial design	non-parametric statistics	factor analysis
Area of Assessment	defect detection	project development	project management
Scope of Technologies	individual technique	development methodology	environment methodology
Factors (F)	code reading functional testing structural testing	Cleanroom development traditional development	SEL environment
Potential Benefit	increase effectiveness of defect detection	increase product quality and process control	better project monitoring and control
Result Statements (S)	characteristic evaluative predictive	characteristic evaluative predictive	characteristic evaluative predictive

3.2.1. Software Testing Strategy Comparison

The software testing strategy analysis is a blocked subject-project study in which 74 individuals applied 3 different testing techniques to 4 different soft-

ware types. The individuals (domain D_1) were selected from the populations of junior, intermediate, and advanced programmers, and the programs tested (domain D_2) were of unit size and were selected from four populations of software types. The programs had a distribution of faults that commonly occur in software. A series of fractional factorial designs was employed in the analysis. Software testing and defect detection are inexact and not very well understood areas of software production. Yet the activities of testing and defect detection are essential to the success of a software project. The three individual techniques (factors F) examined were code reading, functional testing, and structural testing. Result statements (statement strengths S) characterizing, evaluating, and predicting the effect of each of these techniques are intended from the analysis. The major area of benefit from the analysis will be increasing the effectiveness of software testing and defect detection. The goals of this study are to contrast the strategies in three different aspects of software testing: 1) fault detection effectiveness, 2) fault detection cost, and 3) classes of faults detected.

3.2.2. Cleanroom Development Approach Analysis

The Cleanroom development approach analysis is a replicated project study in which 15 small teams (3-person) separately applied two different software development methodologies to build versions of the same small message system: ten teams applied Cleanroom, while five applied a more traditional approach. The individuals (domain D_1) were selected from the populations of junior and

intermediate programmers, and the system built (domain D_2) was a small system selected from the population of small systems of moderate complexity. Non-parametric statistics were applied to contrast the performance of the two development methodologies. The outcome of a software project is largely a function of the development methodology used, and the software community is uncertain which development approaches consistently produce a quality product. The two development methodologies (factors F) examined were Cleanroom software development and a traditional team methodology. The Cleanroom software development approach is intended to produce highly reliable software by integrating formal methods for specification and design, complete offline development, and statistically based testing. Result statements (statement strengths S) characterizing, evaluating, and predicting the effect of the two development methodologies relative to one another are intended from the analysis. The major area of benefit from the analysis will be increasing product quality and development process control. This study analyzes the effect of Cleanroom, relative to a traditional approach, on the delivered product, the software development process, and the developers.

3.2.3. Characteristic Metric Set Study

The characteristic metric set analysis is a multi-project variation study in which one development environment applied its methodology to 6 software projects. The environment (domain D_1) was selected from the population of production environments, and the projects developed (domain D_3) were large sys-

tems selected from the population of large, moderately complex software systems. The quantitative analysis method used was factor analysis. The management of software projects is a challenging and ill-defined task. Better monitoring and control of software projects lead to more successful project management, and possibly higher product requirement conformance and reliability. The environment methodology (factor F) examined was the environment methodology of a NASA Goddard production environment. Result statements (statement strengths S) characterizing, evaluating, and predicting the effect of the particular environment methodology on projects are intended from the analysis. The major area of benefit from the analysis will be increasing the ability to monitor and control software projects. The goals of this study are to 1) develop an approach for customizing a characteristic software metric set to a particular environment; 2) calculate the characteristic metric set for a NASA Goddard environment; and 3) examine the usability of this approach as a management tool.

3.3. Methodology Application

The three analyses described above are intended to advance the understanding of factors that contribute to quality in software development and modification. The next section presents the in-depth analysis for each of the studies, including the goal/question framework, appropriate software metrics, data analysis, and results.

4. Evaluation of Software Technologies: Analysis and Results

The following sections present three studies in which the quantitative methodology described earlier is applied: a blocked subject-project study comparing software testing strategies, a replicated project study characterizing the effect of using the Cleanroom software development approach, and a multi-project variation study to determine a characteristic set of software cost and quality metrics.

4.1. Software Testing Strategy Comparison

The processes of software testing and defect detection continue to challenge the software community. Even though the software testing and defect detection activities are inexact and inadequately understood, they are crucial to the success of a software project. The controlled study presented addresses the uncertainty of how to test software effectively. In this investigation, common testing techniques were applied to different types of software by subjects that had a wide range of professional experience. This work is intended to characterize how testing effectiveness relates to several factors: testing technique, software type, fault type, tester experience, and any interactions among these factors. This examination extends previous work by incorporating different testing techniques and a greater number of persons and programs, while broadening the scope of issues examined and adding statistical significance to the conclusions.

This section describes the testing techniques examined, the investigation goals, the experimental design, operation, analysis, and conclusions.

4.1.1. Testing Techniques

To demonstrate that a particular program actually meets its specifications, professional software developers currently utilize many different testing methods. Before presenting the goals for the empirical study comparing the popular techniques of code reading, functional testing, and structural testing, a description will be given of the testing strategies and their different capabilities (see Figure 4.). In functional testing, which is a "black box" approach [Howden 80], a programmer constructs test data from the program's specification through methods such as equivalence partitioning and boundary value analysis [Myers 79]. The programmer then executes the program and contrasts its actual behavior with that indicated in the specification. In structural testing, which is a "white box" approach [Howden 78, Howden 81], a programmer inspects the source code and then devises and executes test cases based on the percentage of the program's statements or expressions executed (the "test set coverage") [Stucki 77]. The structural coverage criteria used was 100% statement coverage. In code reading by stepwise abstraction, a person identifies prime subprograms in the software, determines their functions, and composes these functions to determine a function for the entire program [Mills 72a, Linger, Mills & Witt 79]. The code reader then compares this derived function and the specifications (the intended function). In order to contrast these various strategies, an empirical study has been conducted using the techniques of code reading, functional testing, and structural testing.

Figure 4. Capabilities of the testing methods.			
	code reading	functional testing	structural testing
view program specification	X	X	X
view source code	X		X
execute program		X	X

4.1.1.1. Investigation Goals

The goals of this study comprise three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. An application of the goal/question/metric paradigm [Basili & Selby 84, Basili & Weiss 84] leads to the framework of goals and questions for this study appearing in Figure 5.

The first goal area is performance oriented and includes a natural first question (I.A): which of the techniques detects the most faults in the programs? The comparison between the techniques is being made across programs, each with a different number of faults. An alternate interpretation would then be to compare the percentage of faults found in the programs (question I.A.1). The number of faults that a technique exposes should also be compared; that is, faults that are made observable but not necessarily observed and reported by a tester (I.A.2). Because of the differences in types of software and in testers' abilities, it is relevant to determine whether the number of faults detected is el-

ther program or programmer dependent (I.B, I.C). Since one technique may find a few more faults than another, it becomes useful to know how much effort that technique requires (II.A). Awareness of what types of software require more effort to test (II.B) and what types of programmer backgrounds require less effort in fault uncovering (II.C) is also quite useful. If one is interested in detecting certain classes of faults, such as in error-based testing [Foster 80, Valdes & Goel 83], it is appropriate to apply a technique sensitive to that particular type (III.A). Classifying the types of faults that are observable yet go unreported could help focus and increase testing effectiveness (III.B).

Figure 5. Structure of goals/subgoals/questions for testing experiment.

I. Fault detection effectiveness

A. For programmers doing unit testing, which of the testing techniques (code reading, functional testing, or structural testing) detects the most faults in programs?

1. Which of the techniques detects the greatest percentage of faults in the programs (the programs each contain a different number of faults)?

2. Which of the techniques exposes the greatest number (or percentage) of program faults (faults that are observable but not necessarily reported)?

B. Is the number of faults observed dependent on software type?

C. Is the number of faults observed dependent on the expertise level of the person testing?

II. Fault detection cost

A. For programmers doing unit testing, which of the testing techniques (code reading, functional testing, or structural testing) detects the

faults at the highest rate (#faults/effort)?

B. Is the fault detection rate dependent on software type?

C. Is the fault detection rate dependent on the expertise level of the person testing?

III. Classes of faults observed

A. For programmers doing unit testing, do the methods tend to capture different classes of faults?

B. What classes of faults are observable but go unreported?

4.1.2. Empirical Study

Admittedly, the goals stated here are quite ambitious. In no way is it implied that this study can definitively answer all of these questions for all environments. It is intended, however, that the statistically significant analysis presented lends insights into their answers and into the merit and appropriateness of each of the techniques. Note that this study compares the individual application of the three testing techniques in order to identify their distinct advantages and disadvantages. This approach is a first step toward proposing a composite testing strategy, which possibly incorporates several testing methods. The following sections describe the empirical study undertaken to pursue these goals and questions, including the selection of subjects, programs, and experimental design, and the overall operation of the study.

4.1.2.1. Iterative Experimentation

The empirical study consisted of three phases. The first and second phases of the study took place at the University of Maryland in the Falls of 1982 and 1983 respectively. The third phase took place at Computer Sciences Corporation (CSC - Silver Spring, MD) and NASA Goddard Space Flight Center (Greenbelt, MD) in the Fall of 1984. The sequential experimentation supported the iterative nature of the learning process, and enabled the initial set of goals and questions to be expanded and resolved by further analysis. The goals were further refined by discussions of the preliminary results [Selby 83, Selby 84]. These three phases enabled the pursuit of result reproducibility across environments having subjects with a wide range of experience.

4.1.2.2. Subject and Program/Fault Selection

A primary consideration in this study was to use a realistic testing environment to assess the effectiveness of these different testing strategies, as opposed to creating a best possible testing situation [Hetzel 76]. Thus, 1) the subjects for the study were chosen to be representative of different levels of expertise, 2) the programs tested correspond to different types of software and reflect common programming style, and 3) the faults in the programs were representative of those frequently occurring in software. Sampling the subjects, programs, and faults in this manner is intended to evaluate the testing methods reasonably, and to facilitate the generalization of the results to other environments.

4.1.2.2.1. Subjects

The three phases of the study incorporated a total of 74 subjects; the individual phases had 29, 13, and 32 subjects respectively. The subjects were selected, based on several criteria, to be representative of three different levels of computer science expertise: advanced, intermediate, and junior. The number of subjects in each level of expertise for the different phases appears in Figure 6.

Figure 6. Expertise levels of subjects.

	Phase			
Level of Expertise	1 (Univ. Md)	2 (Univ. Md)	3 (NASA/CSC)	total
Advanced	0	0	8	8
Intermediate	9	4	11	24
Junior	20	9	13	42
total	29	13	32	74

The 42 subjects in the first two phases of the study were the members of the upper level "Software Design and Development" course at the University of Maryland in the Falls of 1982 and 1983. The individuals were either upper-level computer science majors or graduate students; some were working part-time and all were in good academic standing. The topics of the course included structured programming practices, functional correctness, top-down design, modular specification and design, step-wise refinement, and PDL, in addition to the presentation of the techniques of code reading, functional testing, and structural testing. The references for the testing methods were [Mills 75, Fagan 76, Myers 79, Howden 80], and the lectures were presented by V. R. Basili and F. T. Baker. The subjects from the University of Maryland spanned the inter-

mediate and junior levels of computer science expertise. The assignment of individuals to levels of expertise was based on professional experience and prior academic performance in relevant computer science courses. The individuals in the first and second phases had overall averages of 1.7 (SD = 1.7) and 1.5 (SD = 1.5) years of professional experience. The nine intermediate subjects in the first phase had from 2.8 to 7 years of professional experience (average of 3.9 years, SD = 1.3), and the four in the second phase had from 2.3 to 5.5 years of professional experience (average of 3.2, SD = 1.5). The twenty junior subjects in the first phases and the nine in the second phase both had from 0 to 2 years professional experience (averages of 0.7, SD = 0.6, and 0.8, SD = 0.8, respectively).

The 32 subjects in the third phase of the study were programming professionals from NASA and Computer Sciences Corporation. These individuals were mathematicians, physicists, and engineers that develop ground support software for satellites. They were familiar with all three testing techniques, but had used functional testing primarily. A four hour tutorial on the testing techniques was conducted for the subjects by R. W. Selby. This group of subjects, examined in the third phase of the experiment, spanned all three expertise levels and had an overall average of 10.0 (SD = 5.7) years professional experience. Several criteria were considered in the assignment of subjects to expertise levels, including years of professional experience, degree background, and their manager's suggested assignment. The eight advanced subjects ranged from 9.5 to 20.5 years professional experience (average of 15.0, SD = 4.1). The eleven

intermediate subjects ranged from 3.5 to 17.5 years experience (average of 10.9, SD = 4.9). The thirteen junior subjects ranged from 1.5 to 13.5 years experience (average of 6.1, SD = 4.4).

4.1.2.2.2. Programs

The experimental design enables the distinction of the testing techniques while allowing for the effects of the different programs being tested. The four programs used in the investigation were chosen to be representative of several different types of software. The programs were selected specially for the study and were provided to the subjects for testing; the subjects did not test programs that they had written. All programs were written in a high-level language with which the subjects were familiar. The three programs tested in the CSC/NASA phase were written in FORTRAN, and the programs tested in the University of Maryland phases were written in the Simpl-T structured programming language [Basili & Turner 76]. [2] The four programs tested were P_1) a text processor, P_2) a mathematical plotting routine, P_3) a numeric abstract data type, and P_4) a database maintainer. The programs are summarized in Figure 7. There exists some differentiation in size, and the programs are a realistic size for unit testing. Each of the subjects tested three programs, but a total of four programs was used across the three phases of the study. The programs tested in each of the three phases of the study appear in Figure 8. The specifications for the pro-

[2] Simpl-T is a structured language that supports several string and file handling primitives, in addition to the usual control flow constructs available, for example, in Pascal.

grams appear in Appendix B.1, and their source code appears in Appendix B.2.

Figure 7. The programs tested.

program	source lines	executable statments	cyclomatic complexity	#routines	#faults
P_1 – text formatter	169	55	18	3	9
P_2 – mathematical plotting	145	95	32	8	6
P_3 – numeric data abstraction	147	48	18	9	7
P_4 – database maintainer	365	144	57	7	12

Figure 8. Programs tested in each phase of the analysis.

Program	Phase		
	1 (Univ. Md)	2 (Univ. Md)	3 (NASA/CSC)
P_1 – text formatter	X	X	X
P_2 – mathematical plotting	X	X	
P_3 – numeric data abstraction	X		X
P_4 – database maintainer		X	X

The first program is a text formatting program, which also appeared in [Myers 78]. A version of this program, originally written by [Naur 69] using techniques of program correctness proofs, was analyzed in [Goodenough & Gerhart 75]. The second program is a mathematical plotting routine. This program was written by R. W. Selby, based roughly on a sample program in [Jensen & Wirth 74]. The third program is a numeric data abstraction consisting of a set of list processing utilities. This program was submitted for a class project by a member of an intermediate level programming course at the University of Maryland. [McMullin & Gannon 80]. The fourth program is a maintainer for a

database of bibliographic references. This program was analyzed in [Hetzel 76], and was written by a systems programmer at the University of North Carolina computation center.

Note that the source code for the programs contains no comments. This creates a worst-case situation for the code readers. In an environment where code contained helpful comments, performance of code readers would likely improve, especially if the source code contained as comments the intermediate functions of the program segments. In an environment where the comments were at all suspect, they could then be ignored.

4.1.2.2.3. Faults

The faults contained in the programs tested represent a reasonable distribution of faults that commonly occur in software [Weiss & Basili 85, Basili & Perricone 84]. All the faults in the database maintainer and the numeric abstract data type were made during the actual development of the programs. The other two programs contain a mix of faults made by the original programmer and faults seeded in the code. The programs contained a total of 34 faults; the text formatter had nine, the plotting routine had six, the abstract data type had seven, and the database maintainer had twelve.

4.1.2.2.3.1. Fault Origin

The faults in the text formatter were preserved from the article in which it appeared [Myers 78], except for some of the more controversial ones [Callilau & Rubin 79]. In the mathematical plotter, faults made during program translation

were supplemented by additional representative faults. The faults in the abstract data type were the original ones made by the program's author during the development of the program. The faults in the database maintainer were recorded during the development of the program, and then reinserted into the program. The next section describes a classification of the different types of faults in the programs. Note that this investigation of the fault detecting ability of these techniques involves only those types occurring in the source code, not other types such as those in the requirements or the specifications.

4.1.2.2.3.2. Fault Classification

The faults in the programs are classified according to two different abstract classification schemes [Basili & Perricone 84]. One fault categorization method separates faults of omission from faults of commission. Faults of commission are those faults present as a result of an incorrect segment of existing code. For example, the wrong arithmetic operator is used for a computation in the right-hand-side of an assignment statement. Faults of omission are those faults present as a result of a programmer's forgetting to include some entity in a module. For example, a statement is missing from the code that would assign the proper value to a variable.

A second fault categorization scheme partitions software faults into the six classes of 1) initialization, 2) computation, 3) control, 4) interface, 5) data, and 6) cosmetic. Improperly initializing a data structure constitutes an initialization fault. For example, assigning a variable the wrong value on entry to a module.

Computation faults are those that cause a calculation to evaluate the value for a variable incorrectly. The above example of a wrong arithmetic operator in the right-hand-side of an assignment statement would be a computation fault. A control fault causes the wrong control flow path in a program to be taken for some input. An incorrect predicate in an IF-THEN-ELSE statement would be a control fault. Interface faults result when a module uses and makes assumptions about entities outside the module's local environment. Interface faults would be, for example, passing an incorrect argument to a procedure, or assuming in a module that an array passed as an argument was filled with blanks by the passing routine. A data fault are those that result from the incorrect use of a data structure. For example, incorrectly determining the index for the last element in an array. Finally, cosmetic faults are clerical mistakes when entering the program. A spelling mistake in an error message would be a cosmetic fault.

Interpreting and classifying faults in software is a difficult and inexact task. The categorization process often requires trying to recreate the original programmer's misunderstanding of the problem [Johnson, Draper & Soloway 83]. The above two fault classification schemes attempt to distinguish among different reasons that programmers make faults in software development. They were applied to the faults in the programs in a consistent interpretation; it is certainly possible that another analyst could have interpreted them differently. The separate application of each of the two classification schemes to the faults categorized them in a mutually exclusive and exhaustive manner. Figure 9 displays the distribution of faults in the programs according to these schemes.

Figure 9. Distribution of faults in the programs.			
	Omission	Commission	Total
Initialization	0	2	2
Computation	4	4	8
Control	2	5	7
Interface	2	11	13
Data	2	1	3
Cosmetic	0	1	1
Total	10	24	34

4.1.2.2.3.3. Fault Description

The faults in the programs are described in Figure 10. There have been various efforts to determine a precise counting scheme for "defects" in software [Gloss-Soler 79, IEEE 83]. According to the explanations given, a software "fault" is a specific manifestation in the source code of a programmer "error." For example, due to a misconception or document discrepancy, a programmer commits an "error" (in his/her head) that may result in more than one "fault" in a program. Using this interpretation, software "faults" reflect the correctness, or lack thereof, in a program. The entities examined in this analysis are software faults.

Figure 10. Fault classification and manifestation.

Fault	Program	Omission/ Commission	Class	Description

a	P1	omission	control	a blank is printed before the first word on the first line unless the first word is 30 characters long; in the latter case, a blank line is printed before the first word
b	P1	commission	initialization	the character & (not $) is the new-line character
c	P1	commission	initialization	the line size is 31 characters (not 30); this fault causes the references to the number 30 in the other faults to be actually the number 31
d	P1	commission	interface	since the program pads an empty input buffer with the character "z," it ignores a valid input line that has a "z" as a first character
e	P1	omission	control	successive break characters are not condensed in the output
f	P1	commission	cosmetic	spelling mistake in the error message "*** word to long ***"
g	P1	commission	computation	after detecting a word in the input longer than 30 characters, the message "*** word to long ***" is printed once for every character over 30, and the processing of the text does not terminate
h	P1	omission	interface	after detecting a word in the input longer than 30 characters, the program prints whatever is residing in its output buffer
i	P1	commission	control	after detecting an input line without an end-of-text character, the program erroneously increments its buffer pointer and replaces the first character of the next input line with a "z"
j	P3	commission	interface	routine FIRST returns zero (0) when the list has one element
k	P3	commission	interface	routine ISEMPTY returns true (1) when the list has one element
l	P3	commission	interface	routine DELETEFIRST can not delete the first list element when the list has only one element
m	P3	commission	interface	routine LISTLENGTH returns one less than than the actual length of the list

n	P3	commission	interface	routine ADDFIRST can add more than the specified five elements to the list
o	P3	commission	interface	routine ADDLAST can add more than the specified five elements to the list
p	P3	omission	computation	routine REVERSE does not reverse the list properly when the list has more than one element
q	P4	commission	computation	words greater than or equal to three characters (not strictly greater than) are treated as cross reference keywords
r	P4	commission	interface	since the program uses the key "ZZZ" as an end-of-input sentinel, it does not process a valid record with key "ZZZ" and ignores any following records
s	P4	commission	control	update action add with the error condition "key already in the master file" replaces the existing record; the update record is not ignored
t	P4	commission	control	update action replace with the error condition "key not found in the master file" adds the record; the update record is not ignored
u	P4	omission	data	the number of references and number of words in the dictionary are not checked for overflow
v	P4	omission	computation	two or more update transactions for the same master record give incorrect results
w	P4	commission	interface	keywords longer than 12 characters are truncated and not distinguished
x	P4	commission	control	an update record with column 80 neither an add action "A" nor replace action "R" acts like an add transaction
y	P4	commission	interface	keyword indices appear in reverse alphabetical order
z	P4	omission	interface	no check is made for unique keys in the master file
A	P4	commission	interface	punctuation is made a part of the keyword
B	P4	omission	data	words appearing twice in a title get two cross reference entries
C	P2	commission	computation	the x and y axes are mislabeled

D	P2	omission	computation	points with negative y-values are not processed and do not appear on the graph
E	P2	commission	control	the origin (0,0) appears on the graph regardless of whether it is an input point
F	P2	commission	data	no points can appear on the vertical axis
G	P2	commission	computation	the vertical and horizontal scaling for the pixels are calculated incorrectly, causing some points not to appear in the proper pixel
H	P2	omission	computation	when more than one point would appear in a given pixel, only an asterisk (*) appears, not an appropriate integer

4.1.2.3. Experimental Design

The experimental design applied for each of the three phases of the study was a fractional factorial design [Cochran & Cox 50, Box, Hunter, & Hunter 78]. This experimental design distinguishes among the testing techniques, while allowing for variation in the ability of the particular individual testing or in the program being tested. Figure 11 displays the fractional factorial design appropriate for the third phase of the study. Subject S_1 is in the advanced expertise level, and he structurally tested program P_1, functionally tested program P_3, and code read program P_4. Notice that all of the subjects tested each of the three programs and used each of the three techniques. Of course, no one tests a given program more than once. The design appropriate for the third phase is discussed in the following paragraphs, with the minor differences between this design and the ones applied in the first two phases being discussed at the end of the section.

Figure 11. Fractional Factorial Design.				
		Code Reading	Functional Testing	Structural Testing
		$P_1\ P_3\ P_4$	$P_1\ P_3\ P_4$	$P_1\ P_3\ P_4$
Advanced Subjects	S_1	——X	—X—	X——
	S_2	—X—	X——	——X
	.		. . .	
	.		. . .	
	S_8	X——	——X	—X—
Inter-mediate Subjects	S_9	—X—	X——	——X
	S_{10}	——X	—X—	X——
	.		. . .	
	.		. . .	
	S_{19}	X——	——X	—X—
Junior Subjects	S_{20}	—X—	X——	——X
	S_{21}	X——	——X	—X—
	.		. . .	
	.		. . .	
	S_{32}	——X	—X—	X——

4.1.2.3.1. Independent and Dependent Variables

The experimental design has the three independent variables of testing technique, software type, and level of expertise. For the design appearing in Figure 11, appropriate for the third phase of the study, the three main effects have the following levels:

1) testing technique: code reading, functional testing, and structural testing

2) software type: (P_1) text processing, (P_3) numeric abstract data type, and (P_4) database maintainer

3) level of expertise: advanced, intermediate, and junior

Every combination of these levels occurs in the design. That is, programmers in

all three levels of expertise applied all three testing techniques on all programs. In addition to these three main effects, a factorial analysis of variance (ANOVA) model supports the analysis of interactions among each of these main effects. Thus, the interaction effects of testing technique * software type, testing techniq * expertise level, software type * expertise level, and the three-way interaction of testing technique * software type * expertise level are included in the model. There are several dependent variables examined in the study, including number of faults detected, percentage of faults detected, total fault detection time, and fault detection rate. Observations from the on-line methods of functional and structural testing also had as dependent variables number of computer runs, amount of cpu-time consumed, maximum statement coverage achieved, connect time used, number of faults that were observable from the test data, percentage of faults that were observable from the test data, and percentage of faults observable in the from the test data that were actually observed by the tester.

4.1.2.3.2. Analysis of Variance Model

The three main effects and all the two-way and three-way interactions effects are called fixed effects in this factorial analysis of variance model. The levels of these effects given above represent all levels of interest in the investigation. For example, the effect of testing technique has as particular levels code reading, functional testing, and structural testing; these particular testing techniques are the only ones under comparison in this study. The effect of the par-

ticular subjects that participated in this study requires a little different interpretation. The subjects examined in the study were random samples of programmers from the large population of programmers at each of the levels of expertise. Thus, the effect of the subjects on the various dependent variables is a random variable, and this effect therefore is called a random effect. If the samples examined are truly representative of the population of subjects at each expertise level, the inferences from the analysis can then be generalized across the whole population of subjects at each expertise level, not just across the particular subjects in the sample chosen. Since this analysis of variance model contains both fixed and random effects, it is called a mixed model. The actual ANOVA model for the design appearing in Figure 11 is given below.

$$\Upsilon_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \delta_{kl} + \alpha\beta_{ij} + \alpha\gamma_{ik} + \beta\gamma_{jk} + \alpha\beta\gamma_{ijk} + \epsilon_{ijkl}$$

where

Υ_{ijkl} is the observed response from subject l of experience level k using testing technique i on program j

μ is the overall mean response

α_i is the main effect of testing technique i (i = 1,2,3)

β_j is the main effect of program j (j = 1, 3, 4)

γ_k is the main effect of expertise level k (k = 1, 2, 3)

δ_{kl} is the random effect of subject l within expertise level k, a random variable (l = 1, 2, ..., 32; k = 1, 2, 3)

$\alpha\beta_{ij}$ is the interaction effect of testing technique i with program j (i = 1, 2, 3; j = 1, 3, 4)

$\alpha\gamma_{ik}$ is the interaction effect of testing technique i with expertise level k (i = 1, 2, 3; k = 1, 2, 3)

$\beta\gamma_{jk}$ is the interaction effect of program j with expertise level k (j = 1, 3, 4; k = 1, 2, 3)

$\alpha\beta\gamma_{ijk}$ is the interaction effect of testing technique i with program j with

experience level k (i = 1, 2, 3; j = 1, 3, 4; k = 1, 2, 3)

ϵ_{ijkl} is the experimental error for each observation, a random variable

The F tests of hypotheses on all the fixed effects mentioned above use the error (residual) mean square in the denominator, except for the test of the expertise level effect. The expected mean square for the expertise level effect contains a component for the actual variance of subjects within expertise level. In order to select the appropriate error term for the denominator of the expertise level F test, the mean square for the effect of subjects nested within expertise level is chosen. The parameters for the random effect of subjects within expertise level are assumed to be drawn from a normally distributed random process with mean zero and common variance. The experimental error terms are assumed to have mean zero and common variance.

The fractional factorial design applied in the first two phases of the analysis differed slightly from the one presented above for the third phase.[3] In the third phase of the study, programs P_1, P_3, and P_4 were tested by subjects in three levels of expertise. In both phases one and two, there were only subjects from the levels of intermediate and junior expertise. In phase one, programs P_1, P_3, and P_2 were tested. In phase two, the programs tested were P_1, P_2, and P_4. The only modifications necessary to the above explanation for phases one and two are 1) eliminating the advanced expertise level, 2) changing

[3] Although the data from all the phases can be analyzed together, the number of empty cells resulting from not having all three experience levels and all four programs in all phases limits the number of parameters that can be estimated and causes non-unique Type IV partial sums of squares.

the program P subscripts appropriately, and 3) leaving out the three way interaction term in phase two, because of the reduced number of subjects. In all three of the phases, all subjects used each of the three techniques and tested each of the three programs for that phase. Also, within all three phases, all possible combinations of expertise level, testing techniques, and programs occurred.

The order of presentation of the testing techniques was randomized among the subjects in each level of expertise in each phase of the study. However, the integrity of the results would have suffered if each of the programs in a given phase was tested at different times by different subjects. Note that each of the testing sessions took place on a different day because of the amount of effort required. If different programs would have been tested on different days, any discussion about the programs among subjects between testing sessions would have affected the future performance of others. Therefore, all subjects in a phase tested the same program on the same day. The actual order of program presentation was the order in which the programs are listed in the previous paragraph.

4.1.2.4. Experimental Operation

Each of the three phases were broken into five distinct pieces: training, three testing sessions, and a follow-up session. All groups of subjects were exposed to a similar amount of training on the testing techniques before the study began. As mentioned earlier, the University of Maryland subjects were enrolled in the "Software Design and Development" course, and the NASA/CSC subjects were given a four-hour tutorial. Background information on the subjects

was captured through a questionnaire. Elementary exercises followed by a pretest covering all techniques were administered to all subjects after the training and before the testing sessions. Reasonable effort on the part of the University of Maryland subjects was enforced by their being graded on the work and by their needing to use the techniques in a major class project. Reasonable effort on the part of the NASA/CSC subjects was certain because of their desire for the study's outcome to improve their software testing environment. All subjects groups were judged highly motivated during the study. The subjects were all familiar with the editors, terminals, machines, and the programs' implementation language.

The individuals were requested to use the three testing techniques to the best of their ability. Every subject participated in all three testing sessions of his/her phase, using all techniques but each on a separate program. The individuals using code reading were each given the specification for the program and its source code. They were then asked to apply the methods of code reading by stepwise abstraction to detect discrepancies between the program's abstracted function and the specification. The functional testers were each given a specification and the ability to execute the program. They were asked to perform equivalence partitioning and boundary value analysis to select a set of test data for the program. Then they executed the program on this collection of test data, and inconsistencies between what the program actually performed and what they though the specification said it should perform were noted. The structural testers were given the source code for the program, the ability to exe-

cute it, and a description of the input format for the program. The structural testers were asked to examine the source and generate a set of test cases that cumulatively execute 100% of the program's statements. When the subjects were applying an on-line technique, they generated and executed their own test data; no test data sets were provided. The programs were invoked through a test driver that supported the use the of multiple input data sets. This test driver, unbeknown to the subjects, drained off the input cases submitted to the program for the experimenter's later analysis; the programs could only be accessed through a test driver.

A structural coverage tool calculated the actual statement coverage of the test set and which statements were left unexecuted for the structural testers. After the structural testers generated a collection of test data that met (or almost met) the 100% coverage criteria, no further execution of the program or reference to the source code was allowed. They retained the program's output from the test cases they had generated. These testers were then provided with the program's specification. Now that they knew what the program was intended to do, they were asked to contrast the program's specification with the behavior of the program on the test data they derived. This scenario for the structural testers was necessary so that "observed" faults could be compared.

At the end of each of the testing sessions, the subjects were asked to give a reasonable estimate of the amount of time spent detecting faults with a given testing technique. The University of Maryland subjects were assured that this had nothing to with the grading of the work. There seemed to be little incen-

tive for the subjects in any of the groups not to be truthful. At the completion of each testing session, the NASA/CSC subjects were also asked what percentage of the faults in the program that they thought were uncovered. After all three testing sessions in a given phase were completed, the subjects were requested to critique and evaluate the three testing techniques regarding their understandability, naturalness, and effectiveness. The University of Maryland subjects submitted a written critique, while a two hour debriefing forum was conducted for the NASA/CSC individuals. In addition to obtaining the impressions of the individuals, these follow-up procedures gave an understanding of how well the subjects were comprehending and applying the methods. These final sessions also afforded the participants an opportunity to comment on any particular problems they had with the techniques or in applying them to the given programs.

4.1.3. Data Analysis

The analysis of the data collected from the various phases of the experiment is presented according to the goal and question framework discussed earlier.

4.1.3.1. Fault Detection Effectiveness

The first goal area addresses the fault detection effectiveness of each of the techniques. Figure 12 presents a summary of the measures that were examined to pursue this goal area. A brief description of each measure is as follows - (*) means only relevant for on-line testing. a) # Faults detected - the number of

faults detected by a subject applying a given testing technique on a given program. b) % Faults detected – the percentage of a program's faults that a subject detected by applying a testing technique to the program. c) # Faults observable (*) – the number of faults that were observable from the program's behavior given the input data submitted. d) % Faults observable (*) – the percentage of a program's faults that were observable from the program's behavior given the input data submitted. e) % Detected/observable (*) – the percentage of faults observable from the program's behavior on the given input set that were actually observed by a subject. f) % Faults felt found – a subject's estimate of the percentage of a program's faults that he/she thought were detected by his/her testing. g) Maximum statement coverage (*) – the maximum percentage of a program's statements that were executed in a set of test cases.

4.1.3.1.1. Data Distributions

The actual distribution of the number of faults observed by the subjects appears in Figure 13, broken down by phase. From Figures 12 and 13, the large variation in performance among the subjects is clearly seen. The mean number of faults detected by the subjects is displayed in Figure 14, broken down by technique, program, expertise level, and phase.

Figure 12.
Overall summary of detection effectiveness data.
Note: some data pertain to only on-line techniques (*), and some data were collected only in certain phases.

Phase	#Subj.	Measure	Mean	SD	Min.	Max.
1	29	# Faults detected	3.94	1.82	0.00	7.00
1	29	% Faults detected	54.78	26.11	0.00	100.00
1	29(*)	# Faults observable	5.38	1.51	3.00	8.00
1	29(*)	% Faults observable	74.59	20.54	33.33	100.00
1	29(*)	% Detected/observable	70.99	24.01	0.00	100.00
2	13	# Faults detected	3.28	1.96	0.00	7.00
2	13	% Faults detected	39.53	27.25	0.00	100.00
3	32	# Faults detected	4.27	1.86	0.00	8.00
3	32	% Faults detected	49.82	27.44	0.00	100.00
3	32	% Faults felt found	75.10	24.07	0.00	100.00
3	32(*)	# Faults observable	5.61	1.52	3.00	9.00
3	32(*)	% Faults observable	62.11	18.36	25.00	100.00
3	32(*)	% Detected/observable	69.67	27.14	0.00	100.00
3	32(*)	Max. % stmt. covered	97.02	7.83	46.00	100.00
Ave	74	# Faults detected	3.97	1.88	0.00	8.00
Ave	74	% Faults detected	49.96	27.29	0.00	100.00
Ave	61(*)	# Faults observable	5.5	1.5	3.00	9.00
Ave	61(*)	% Faults observable	68.0	20.3	25.0	100.0
Ave	61(*)	% Detected/observable	70.3	25.6	0.0	100.0

Figure 13. Distribution of the number of faults detected broken down by phase. Key: code readers (C), functional testers (F), and structural testers (S).

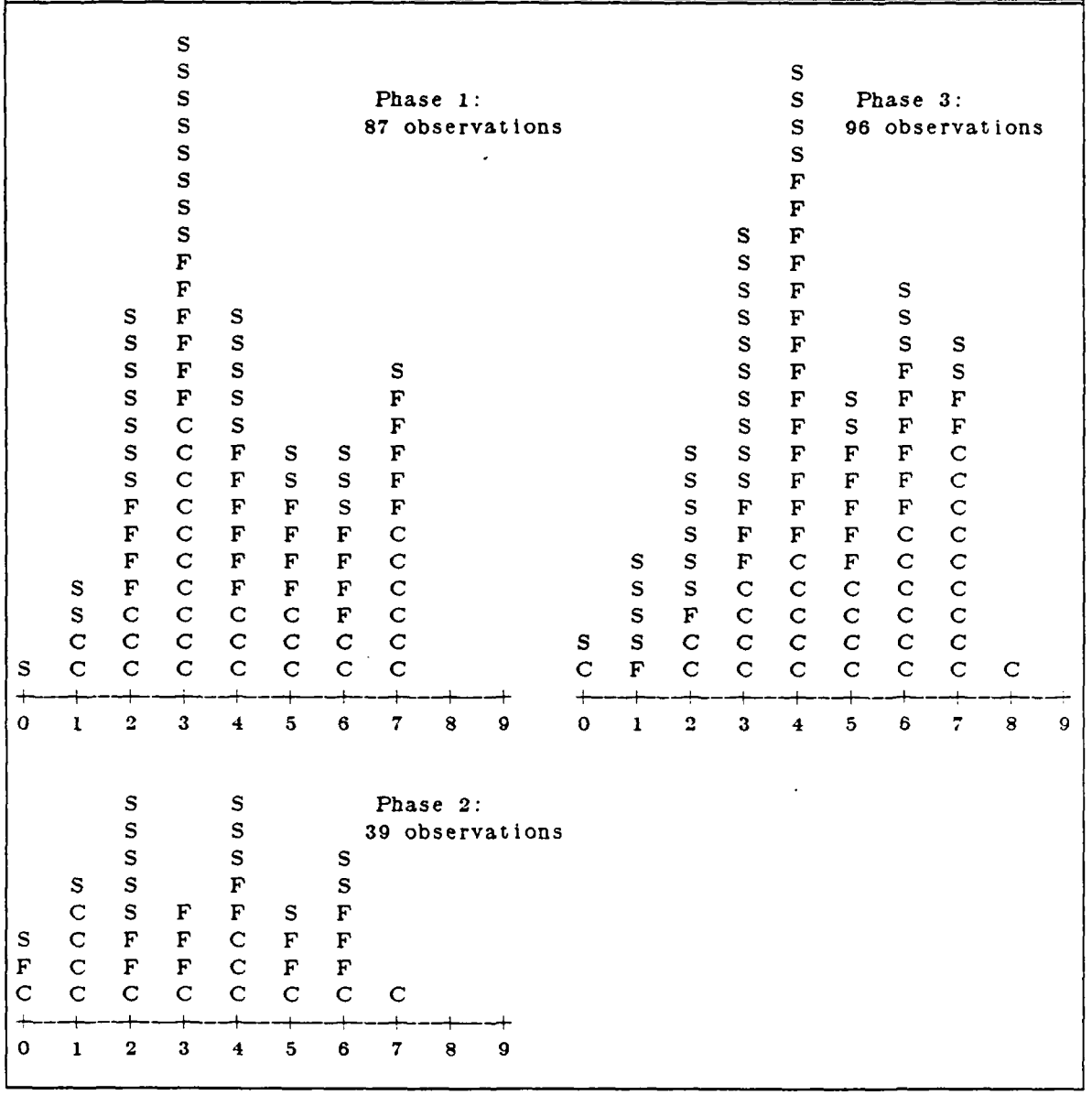


Figure 14.
Overall summary for number of faults detected.

		Phase		
		1	2	3
Effect	Level	Mean(SD)	Mean(SD)	Mean(SD)
Technique	Reading	4.10 (1.93)	3.00 (2.20)	5.09 (1.92)
	Functional	4.45 (1.70)	3.77 (1.83)	4.47 (1.34)
	Structural	3.28 (1.67)	3.08 (1.89)	3.25 (1.80)
Program	Formatter	4.07 (1.62)	3.23 (2.20)	4.19 (1.73)
	Plotter	3.48 (1.45)	3.31 (1.97)	. (.)
	Data type	4.28 (2.25)	. (.)	5.22 (1.75)
	Database	. (.)	3.31 (1.84)	3.41 (1.66)
Expertise	Junior	3.88 (1.89)	3.04 (2.07)	3.90 (1.83)
	Intermed.	4.07 (1.69)	3.83 (1.64)	4.18 (1.99)
	Advanced	. (.)	. (.)	5.00 (1.53)

4.1.3.1.2. Number of Faults Detected

The first question under this goal area asks which of the testing techniques detected the most faults in the programs. The overall F-test of the techniques detecting an equal number of faults in the programs is rejected in the first and third phases of the study ($\alpha < .024$ and $\alpha < .0001$, respectively; not rejected in phase two, $\alpha > .05$). Recall that the phase three data was collected from 32 NASA/CSC subjects, and the phase one data was from 29 University of Maryland subjects. With the phase three data, the contrast of "reading − 0.5 * (functional + structural)" estimates that the technique of code reading by stepwise abstraction detected 1.24 more faults per program than did either of the

other techniques ($\alpha < .0001$, c.l. 0.73 – 1.75).[4] Note that code reading performed well even though the professional subjects' primary experience was with functional testing. Also with the phase three data, the contrast of "functional – structural" estimates that the technique of functional testing detected 1.11 more faults per program than did structural testing ($\alpha < .0007$, c.l. 0.52 – 1.70). In the phase one data, the contrast of "0.5 * (reading + functional) – structural" estimates that the technique of structural testing detected 1.00 fault less per program than did either reading or functional testing ($\alpha < .0065$, c.l. 0.31 – 1.69). In the phase one data, the contrast of "reading – functional" was not statistically different from zero ($\alpha > .05$). The poor performance of structural testing across the phases suggests the inadequacy of using statement coverage criteria. The above pairs of contrasts were chosen because they are linearly independent.

4.1.3.1.3. Percentage of Faults Detected

Since the programs tested each had a different number of faults, a question in the earlier goal/question framework asks which technique detected the greatest percentage of faults in the programs. The order of performance of the techniques is the same as above when the percentage of the programs' faults detected are compared. The overall F-tests for phases one and three were rejected as before ($\alpha < .037$ and $\alpha < .0001$ respectively; not rejected in phase two.

[4] The probably of Type I error is reported, the probability of erroneously rejecting the null hypothesis. The abbreviation "c.l." stands for 95% confidence interval.

$\alpha > .05$). Applying the same contrasts as above: a) in phase three, reading detected 16.0% more faults per program than did the other techniques ($\alpha < .0001$, c.i. 9.9 – 22.1), and functional detected 11.2% more faults than did structural ($\alpha < .003$, c.i. 4.1 – 18.3); b) in phase one, structural detected 13.2% fewer of a program's faults than did the other methods ($\alpha < .011$, c.i. 3.5 – 22.9), and reading and functional were not statistically different as before.

4.1.3.1.4. Dependence on Software Type

Another question in this goal area queries whether the number or percentage of faults detected depends on the program being tested. The overall F-test that the number of faults detected is not program dependent is rejected only in the phase three data ($\alpha < .0001$). Applying Tukey's multiple comparison on the phase three data reveals that the most faults were detected in the abstract data type, the second most in the text formatter, and the least number of faults were found in the database maintainer (simultaneous $\alpha < .05$). When the percentage of faults found in a program is considered, however, the overall F-tests for the three phases are all rejected ($\alpha < .027$, $\alpha < .01$, and $\alpha < .0001$ in respective order). Tukey's multiple comparison yields the following orderings on the programs (all simultaneous $\alpha < .05$). In the phase one data, the ordering was (data type $\simeq$ plotter) $>$ text formatter; that is, a higher percentage of faults were detected in either the abstract data type or the plotter than were found in the text formatter; there was no difference between the abstract data type and the plotter in the percentage found. In the phase two data, the ordering of percen-

tage of faults detected was plotter > (text formatter ≃ database maintainer). In the phase three data, the ordering of percentage of faults found in the programs was the same as the number of faults found, abstract data type > text formatter > database maintainer. Summarizing the effect of the type of software on the percentage of faults observed: 1) the programs with the highest percentage of their faults detected were the abstract data type and the mathematical plotter, the percentage detected between these two was not statistically different; 2) the programs with the lowest percentage of their faults detected were the text formatter and the database maintainer; the percentage detected between these two was not statistically different in the phase two data, but a higher percentage of faults in the text formatter was detected in the phase three data.

4.1.3.1.5. Observable vs. Observed Faults

One evaluation criteria of the success of a software testing session is the number of faults detected. An evaluation criteria of the particular test data generated, however, is the ability of the test data to reveal faults in the program. A test data set's ability to uncover faults in a program can be measured by the number or percentage of a program's faults that are made observable from execution on that input. Distinguishing the faults observable in a program from the faults actually observed by a tester highlights the differences in the activities of test data generation and program behavior examination. As shown in Figure 11, the average number of the programs' faults observable was 68.0%

when individuals were either functional testing or structurally testing. Of course, with a nonexecution-based technique such as code reading, 100% of the faults are observable. Test data generated by subjects using the technique of functional testing resulted in 1.4 more observable faults ($\alpha<.0002$, c.i. 0.79 – 2.01) than did the use of structural testing in phase one of the study; the percentage difference of functional over structural was estimated at 20.0% ($\alpha<.0002$, c.i. 11.2 – 28.8). The techniques did not differ in these two measures in the third phase of the study. However, just considering the faults that were observable from the submitted test data, functional testers detected 18.5% more of these observable faults than did structural testers in the phase three data ($\alpha<.0016$, c.i. 8.9 – 28.1); they did not differ in the phase one data. Note that all faults in the programs could be observed in the programs' output given the proper input data. When using the on-line techniques of functional and structural testing, subjects detected 70.3% of the faults observable in the program's output. In order to conduct a successful testing session, faults in a program must be both revealed and subsequently observed.

4.1.3.1.6. Dependence on Program Coverage

Another measure of the ability of a test set to reveal a program's faults is the percentage of a program's statements that are executed by the test set. The average maximum statement coverage achieved by the functional and structural testers was 97.0%. The maximum statement coverage from the submitted test data was not statistically different between the functional and structural testers

($\alpha > .05$). Also, there was no correlation between maximum statement coverage achieved and either number or percentage of faults found ($\alpha > .05$).

4.1.3.1.7. Dependence on Programmer Expertise

A final question in this goal area concerns the contribution of programmer expertise to fault detection effectiveness. In the phase three data from the NASA/CSC professional environment, subjects of advanced expertise detected more faults than did either the subjects of intermediate or junior expertise ($\alpha < .05$). When the percentage of faults detected is compared, however, the advanced subjects performed better than the junior subjects ($\alpha < .05$), but were not statistically different from the intermediate subjects ($\alpha > .05$). The intermediate and junior subjects were not statistically different in any of the three phases of the study in terms of number or percentage faults observed. When several subject background attributes were correlated with the number of faults found, total years of professional experience had a minor relationship (Pearson $R = .22$, $\alpha < .05$). Correspondence of performance with background aspects was examined across all observations, and within each of the phases, including previous academic performance for the University of Maryland subjects. Other than the above, no relationships were found.

4.1.3.1.8. Accuracy of Self-Estimates

Recall that the NASA/CSC subjects in the phase three data estimated, at the completion of a testing session, the percentage of a program's faults they thought they had uncovered. This estimation of the number of faults un-

covered correlated reasonably well with the actual percentage of faults detected ($R = .57$, $\alpha < .0001$). Investigating further, individuals using the different techniques were able to give better estimates: code readers gave the best estimates ($R = .79$, $\alpha < .0001$), structural testers gave the second best estimates ($R = .57$, $\alpha < .0007$), and functional testers gave the worst estimates (no correlation, $\alpha > .05$). This last observation suggests that the code readers were more certain of the effectiveness they had in revealing faults in the programs.

4.1.3.1.9. Dependence on Interactions

There were few significant interactions between the main effects of testing technique, program, and expertise level. In the phase two data, there was an interaction between testing technique and program in both the number and percentage of faults found ($\alpha < .0013$, $\alpha < .0014$ respectively). The effectiveness of code reading increased on the text formatter. In the phase three data, there was a slight three-way interaction between testing technique, program, and expertise level for both the number and percentage of faults found ($\alpha < .05$, $\alpha < .04$ respectively).

4.1.3.1.10. Summary of Fault Detection Effectiveness

Summarizing the major results of the comparison of fault detection effectiveness: 1) In the phase three data, code reading detected a greater number and percentage of faults than the other methods, with functional detecting more than structural; 2) In the phase one data, code reading and functional were equally effective, while structural was inferior to both – there were no differences

among the three techniques in phase two; 3) the number of faults observed depends on the type of software: the most faults were detected in the abstract data type and the mathematical plotter, the second most in the text formatter, and (in the case of the phase three data) the least were found in the database maintainer; 4) functionally generated test data revealed more observable faults than did structurally generated test data in phase one, but not in phase three; 5) subjects of intermediate and junior expertise were equally effective in detecting faults, while advanced subjects found a greater number of faults than did either group; and 6) self-estimates of faults detected were most accurate from subjects applying code reading, followed by those doing structural testing, with estimates from persons functionally testing having no relationship.

4.1.3.2. Fault Detection Cost

The second goal area examines the fault detection cost of each of the techniques. Figure 15 presents a summary of the measures that were examined to investigate this goal area. A brief description of each measure is as follows – (*) means only relevant for on-line testing. a) # Faults / hour – the number of faults detected by a subject applying a given technique normalized by the effort in hours required, called the fault detection rate. b) Detection time – the total number of hours that a subject spent in testing a program using a technique. c) Cpu-time (*) – the cpu-time in seconds used during the testing session. d) Normalized cpu-time (*) – the cpu-time in seconds used during the testing session.

normalized by a factor for machine speed.[5] e) Connect time (*) – the number of minutes that a individual spent on-line while testing a program. f) # Program runs (*) – the number of executions of the program test driver; note that the driver supported multiple sets of input data. All of the on-line statistics were monitored by the operating systems of the machines.

4.1.3.2.1. Data Distributions

The actual distribution of the fault detection rates for the subjects appears in Figure 16, broken down by phase. Once again, note the many-to-one differential in subject performance. Figure 17 displays the mean fault detection rate for the subjects, broken down by technique, program, expertise level, and phase.

[5] In the phase three data, testing was done on both a VAX 11/780 and an IBM 4341. As suggested by benchmark comparisons [Church 84], the VAX cpu-times were divided by 1.6 and the IBM cpu-times were divided by 0.9.

Figure 15.
Overall summary of fault detection cost data.
Note: some data pertain to only on-line techniques (*), and some data were collected only in certain phases.

Phase	#Subj.	Measure	Mean	SD	Min.	Max.
1	29	# Faults / hour	1.63	1.28	0.00	7.00
1	29	Detection time (hrs)	3.33	2.09	0.75	10.00
2	13	# Faults / hour	0.99	0.81	0.00	3.00
2	13	Detection time (hrs)	4.70	3.02	1.00	14.00
3	32	# Faults / hour	2.33	2.28	0.00	14.00
3	32	Detection time (hrs)	2.75	1.57	0.50	7.25
3	32(*)	Cpu-time (sec)	45.2	56.1	3.0	283.0
3	32(*)	Cpu-time (sec; norm.)	38.5	51.7	2.9	314.4
3	32(*)	Connect time (min)	65.83	50.21	3.50	214.00
3	32(*)	# program runs	5.45	5.00	1.00	24.00
Ave	74	# Faults / hour	1.82	1.80	0.00	14.00
Ave	74	Detection time (hrs)	3.32	2.19	0.50	14.00

Figure 16. Distribution of the fault detection rate (#faults detected per hour) broken down by phase. Key: code readers (C), functional testers (F), and structural testers (S).

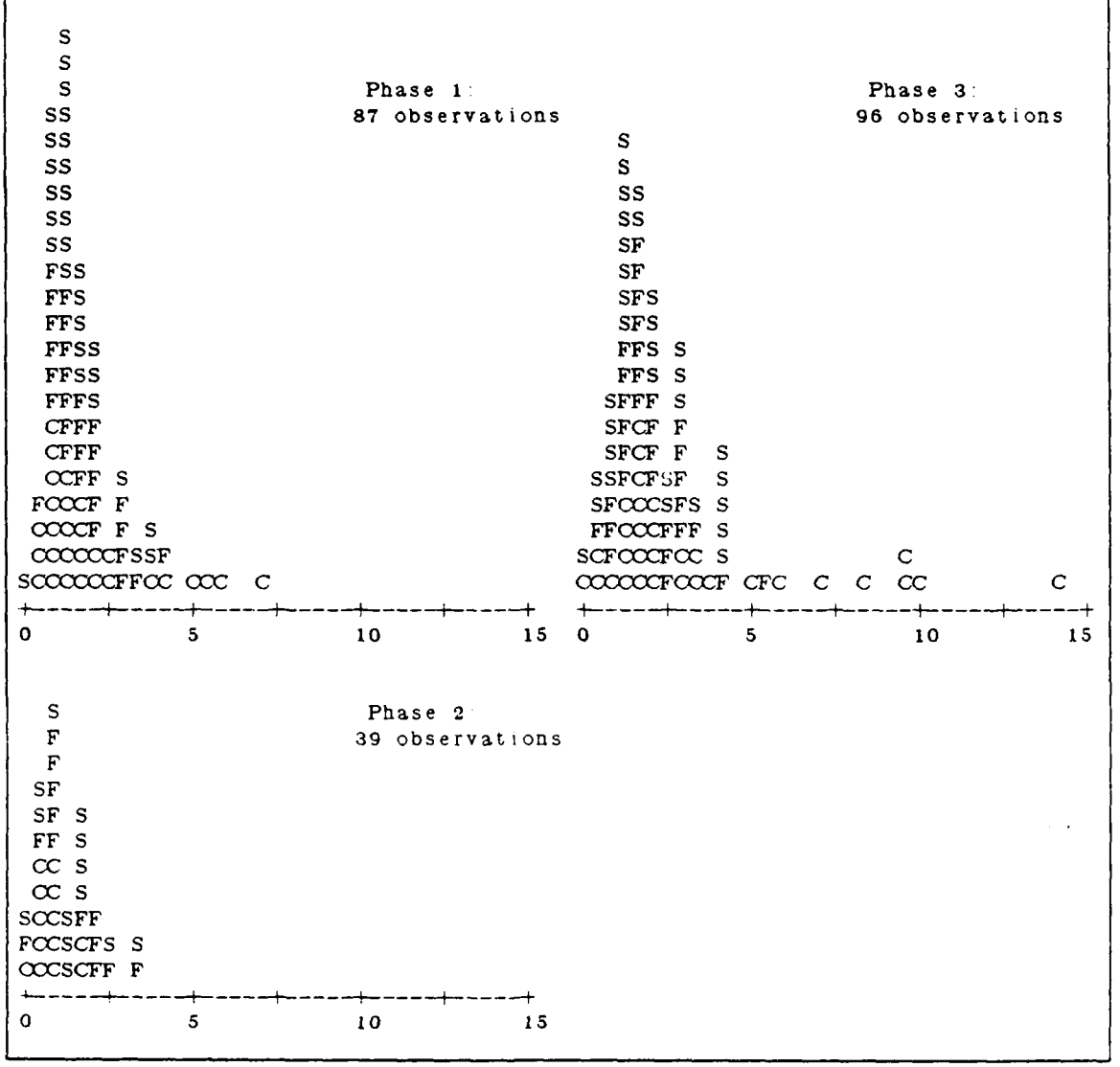


Figure 17.
Overall summary for fault detection rate (# faults detected per hour).

		Phase		
		1	2	3
Effect	Level	Mean(SD)	Mean(SD)	Mean(SD)
Technique	Reading	1.90 (1.83)	0.56 (0.46)	3.33 (3.42)
	Functional	1.58 (0.90)	1.22 (0.91)	1.84 (1.06)
	Structural	1.40 (0.87)	1.18 (0.84)	1.82 (1.24)
Program	Formatter	1.60 (1.39)	0.98 (0.67)	2.15 (1.10)
	Plotter	1.19 (0.83)	0.92 (0.71)	. (.)
	Data type	2.09 (1.42)	. (.)	3.70 (3.26)
	Database	. (.)	1.05 (1.04)	1.14 (0.79)
Expertise	Junior	1.36 (0.97)	1.00 (0.85)	2.14 (2.48)
	Intermed.	2.22 (1.66)	0.96 (0.74)	2.53 (2.48)
	Advanced	. (.)	. (.)	2.36 (1.61)

4.1.3.2.2. Fault Detection Rate and Total Time

The first question in this goal area asks which testing technique had the highest fault detection rate. The overall F-test of the techniques' having the same fault detection rate was rejected in the phase three data ($\alpha < .0014$), but not in the other two phases ($\alpha > .05$). As before, the two contrasts of "reading – 0.5 * (functional + structural)" and "functional – structural" were examined to detect differences among the techniques. The technique of code reading was estimated at detecting 1.49 more faults per hour than did the other techniques in the phase three data ($\alpha < .0003$, c.i. 0.75 – 2.23). The techniques of functional and structural testing were not statistically different ($\alpha > .05$). Comparing the total time spent in fault detection, the techniques were not statistically different

in the phase two and three data; the overall F-test for the phase one data was rejected ($\alpha < .013$). In the phase one data, structural testers spent an estimated 1.08 hours less testing than did the other techniques ($\alpha < .004$, c.i. 0.39 - 1.78), while code readers were not statistically different from functional testers. Recall that in phase one, the structural testers observed both a lower number and percentage of the programs' faults than did the other techniques.

4.1.3.2.3. Dependence on Software Type

Another question in this area focuses on how fault detection rate depends on software type. The overall F-test that the detection rate is the same for the programs is rejected in the phase one and phase three data ($\alpha < .01$ and $\alpha < .0001$ respectively); the detection rate among the programs was not statistically different in phase two. Applying Tukey's multiple comparisons on the phase one data finds that the fault detection rate was greater on the abstract data type than on the plotter, while there was no difference either between the abstract data type and the text formatter or between the text formatter and the plotter (simultaneous $\alpha < .05$). In the phase three data, the fault detection rate was higher in the abstract data type than it was for the text formatter and the database maintainer, with the text formatter and the database maintainer not being statistically different (simultaneous $\alpha < .05$). The overall effort spent in fault detection was different among the programs in phases one and three ($\alpha < .012$ and $\alpha < .0001$ respectively), while there was no difference in phase two. In phase one, more effort was spent testing the plotter than the abstract data

type, while there was no statistical difference either between the plotter and the text formatter or between the text formatter and the abstract data type (simultaneous $\alpha<.05$). In phase three, more time was spent testing the database maintainer than was spent on either the text formatter or on the abstract data type, with the text formatter not differing from the abstract data type (simultaneous $\alpha<.05$). Summarizing the dependence of fault detection cost on software type, 1) the abstract data type had a higher detection rate and less total detection effort than did either the plotter or the database maintainer, the latter two were not different in either detection rate or total detection time; 2) the text formatter and the plotter did not differ in fault detection rate or total detection effort; 3) the text formatter and the database maintainer did not differ in fault detection rate overall and did not differ in total detection effort in phase two, but the database maintainer had a higher total detection effort in phase three; 4) the text formatter and the abstract data type did not differ in total detection effort overall and did not differ in fault detection rate in phase one, but the abstract data type had a higher detection rate in phase three.

4.1.3.2.4. Computer Costs

In addition to the effort spent by individuals in software testing, on-line methods incur machine costs. The machine cost measures of cpu-time, connect time, and the number of runs were compared across the on-line techniques of functional and structural testing in phase three of the study. A nonexecution-based technique such as code reading, of course, incurs no machine time costs.

When the machine speeds are normalized (see measure definitions above), the technique of functional testing used 26.0 more seconds of cpu-time than did the technique of structural testing ($\alpha<.016$, c.i. 7.0 – 45.0). The estimate of the difference is 29.6 seconds when the cpu-times are not normalized ($\alpha<.012$, c.i. 9.0 – 50.2). Individuals using functional testing used 28.4 more minutes of connect time than did those using structural testing ($\alpha<.004$, c.i. 11.7 – 45.1). The number of computer runs of a program's test driver was not different between the two techniques ($\alpha>.05$). These results suggest that individuals using functional testing spent more time on-line and used more cpu-time per computer run than did those structurally testing.

4.1.3.2.5. Dependence on Programmer Expertise

The relation of programmer expertise to cost of fault detection is another question in this goal section. The expertise level of the subjects had no relation to the fault detection rate in phases two and three ($\alpha>.05$ for both F-tests). Recall that phase three of the study used 32 professional subjects with all three levels of computer science expertise. In phase one, however, the intermediate subjects detected faults at a faster rate than did the junior subjects ($\alpha<.005$). The total effort spent in fault detection was not different among the expertise levels in any of the phases ($\alpha>.05$ for all three F-tests). When all 74 subjects are considered, years of professional experience correlates positively with fault detection rate ($R = .41$, $\alpha<.0002$) and correlates slightly negatively with total detection time ($R = -.25$, $\alpha<.03$). These last two observations suggest that

persons with more years of professional experience detected the faults faster and spent less total time doing so. Several other subject background measures showed no relationship with fault detection rate or total detection time ($\alpha < .05$). Background measures were examined across all subjects and within the groups of NASA/CSC subjects and University of Maryland subjects.

4.1.3.2.6. Dependence on Interactions

There were few significant interactions between the main effects of testing technique, program, and expertise level. There was an interaction between testing technique and software type in terms of fault detection rate and total detection cost for the phase three data ($\alpha < .003$ and $\alpha < .007$ respectively). Subjects using code reading on the abstract data type had an increased fault detection rate and a decreased total detection time.

4.1.3.2.7. Relationships Between Fault Detection Effectiveness and Cost

There were several correlations between fault detection cost measures and performance measures. Fault detection rate correlated overall with number of faults detected ($R = .48$, $\alpha < .0001$), percentage of faults found ($R = .48$, $\alpha < .0001$), and total detection time ($R = -.53$, $\alpha < .0001$), but not with normalized cpu-time, raw cpu-time, connect time, or number of computer runs ($\alpha > .05$). Total detection time correlated with normalized cpu-time ($R = .36$, $\alpha < .04$) and raw cpu-time ($R = .37$, $\alpha < .04$), but not with connect time, number of runs, number of faults detected, or percentage of faults detected.

The number of faults detected in the programs correlated with the amount of machine resources used: normalized cpu-time ($R = .47$, $\alpha < .007$), raw cpu-time ($R = .52$, $\alpha < .002$), and connect time ($R = .49$, $\alpha < .003$), but not with the number of computer runs ($\alpha > .05$). The correlations for percentage of faults detected with machine resources used were similar. Although most of these correlations are minor, they suggest that 1) the higher the fault detection rate, the more faults found and the less time spent in fault detection; 2) fault detection rate had no relationship with use of machine resources; 3) spending more time in detecting faults had no relationship with the amount of faults detected; and 4) the more cpu-time and connect time used, the more faults found.

4.1.3.2.8. Summary of Fault Detection Cost

Summarizing the major results of the comparison of fault detection cost: 1) in the phase three data, code reading had a higher fault detection rate than the other methods, with no difference between functional testing and structural testing; 2) in the phase one and two data, the three techniques were not different in fault detection rate; 3) in the phase two and three data, total detection effort was not different among the techniques, but in phase one less effort was spent for structural testing than for the other techniques, while reading and functional were not different; 4) fault detection rate and total effort in detection depended on the type of software: the abstract data type had the highest detection rate and lowest total detection effort, the plotter and the database maintainer had the lowest detection rate and the highest total detection effort, and the text for-

matter was somewhere in between depending on the phase; 5) functional testing used more cpu-time and connect time than did structural testing, but they were not different in the number of runs; 6) in phases two and three, subjects across expertise levels were not different in fault detection rate or total detection time, in phase one intermediate subjects had a higher detection rate; and 7) there was a moderate correlation between fault detection rate and years of professional experience across all subjects.

4.1.3.3. Characterization of Faults Detected

The third goal area focuses on determining what classes of faults are detected by the different techniques. In the earlier section on the faults in the software, the faults were characterized by two different classification schemes: omission or commission, and initialization, control, data, computation, interface, or cosmetic. The faults detected across all three study phases are broken down by the two fault classification schemes in Figure 18. The entries in the figure are the average percentage (with standard deviations) of faults in a given class observed when a particular technique was being used. Note that when a subject tested a program that had no faults in a given class, he/she was excluded from the calculation of this average.

Figure 18. Characterization of the faults detected.				
	Code Reading	Functional Testing	Structural Testing	Overall
Omission	55.6 (40.1)	61.0 (39.5)	39.2 (41.6)	52.0 (41.3)
Commission	54.3 (32.1)	53.5 (25.4)	44.3 (26.6)	50.7 (28.4)
Total	54.1 (29.2)	54.6 (24.5)	41.2 (26.1)	50.0 (27.3)
Initial.	64.6 (40.3)	75.0 (36.1)	46.2 (39.8)	61.5 (40.2)
Control	42.8 (36.6)	66.7 (34.9)	48.8 (36.5)	52.8 (37.2)
Data	20.7 (36.6)	28.3 (44.9)	26.8 (41.9)	25.3 (41.0)
Computat.	70.9 (37.0)	64.2 (40.8)	58.8 (43.5)	64.6 (40.6)
Interface	46.7 (38.5)	30.7 (33.5)	24.6 (29.4)	34.1 (35.1)
Cosmetic	16.7 (38.1)	8.3 (28.2)	7.7 (27.2)	10.8 (31.3)
Total	54.1 (29.2)	54.6 (24.5)	41.2 (26.1)	50.0 (27.3)

4.1.3.3.1. Omission vs. Commission Classification

When the faults are partitioned according to the omission/commission scheme, there is a distinction among the techniques. Both code readers and functional testers observed more omission faults than did structural testers ($\alpha < .001$), with code readers and functional testers not being different ($\alpha > .05$). Since a fault of omission occurs as a result of some segment of code being left out, you would not expect structurally generated test data to find such faults. In fact, 44% of the subjects applying structural testing found zero faults of omission when testing a program.

4.1.3.3.2. Six-Part Fault Classification

When the faults are divided according to the second fault classification scheme, several differences are apparent. Both code reading and functional testing found more initialization faults than did structural testing ($\alpha < .05$), with

code reading and functional testing not being different ($\alpha > .05$). Code reading detected more interface faults than did either of the other methods ($\alpha < .01$), with no difference between functional and structural testing ($\alpha > .05$). This suggests that the code reading process of abstracting and composing program functions across modules must be an effective technique for finding interface faults. Functional testing detected more control faults than did either of the other methods ($\alpha < .01$), with code reading and structural testing not being different ($\alpha > .05$). Recall that the structural test data generation criteria examined is based on determining the execution paths in a program and deriving test data that execute 100% of the program's statements. One would expect that more control path faults would be found by such a technique. However, structural testing did not do as well as functional testing in this fault class. The technique of code reading found more computation faults than did structural testing ($\alpha < .05$), with functional testing not being different from either of the other two methods ($\alpha > .05$). The three techniques were not statistically different in the percentage of faults they detected in either the data or cosmetic fault classes ($\alpha > .05$ for both).

4.1.3.3.3. Observable Fault Classification

Figure 19 displays the average percentage (with standard deviations) of faults from each class that were observable from the test data submitted, yet were not reported by the tester.[6] The two on-line techniques of functional and

[6] The standard deviations presented in the figure are high because of the several instances in which all observable faults were reported.

structural testing were not different in any of the faults classes ($\alpha > .05$). Note that there was only one fault in the cosmetic class.

Figure 19. Characterization of the faults observable, but not reported.

	Functional Testing	Structural Testing	Overall
Omission	15.7 (25.4)	21.3 (31.8)	18.5 (28.8)
Commission	19.1 (20.0)	20.1 (16.6)	19.6 (18.3)
Total	18.1 (17.8)	19.9 (16.8)	19.0 (17.3)
Initial.	5.0 (15.4)	14.3 (32.2)	9.8 (25.5)
Control	20.3 (30.6)	21.1 (31.4)	20.7 (30.8)
Data	28.6 (43.5)	7.5 (24.5)	18.3 (36.7)
Computat.	16.0 (31.3)	20.1 (37.6)	18.0 (34.5)
Interface	16.1 (20.0)	20.3 (21.5)	18.2 (20.8)
Cosmetic	60.0 (50.3)	85.7 (35.9)	73.2 (44.9)
Total	18.1 (17.8)	19.9 (16.8)	19.0 (17.3)

4.1.3.3.4. Summary of Characterization of Faults Detected

Summarizing the major results of the comparison of classes of faults detected: 1) code reading and functional testing both detected more omission faults and initialization faults than did structural testing; 2) code reading detected more interface faults than did the other methods; 3) functional testing detected more control faults than did the other methods; 4) code reading detected more computation faults than did structural testing; and 5) the on-line techniques of functional and structural testing were not different in any classes of faults observable but not reported.

4.1.4. Conclusions

This study compares the strategies of code reading, functional testing, and structural testing across three data sets in three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Each of the three testing techniques showed merit in this evaluation. The investigation was intended to compare the different testing strategies in a representative testing situation, using programmers with a wide range of experience, different software types, and common software faults.

The major results of this study are 1) with the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, with functional testing detecting more faults than did structural testing, and with functional and structural testing not differing in fault detection rate; 2) in one UoM subject group, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in the other UoM subject group there was no difference among the techniques; 3) with the UoM subjects, the three techniques were not different in fault detection rate; 4) number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested; 5) code reading detected more interface faults than did the other methods; and 6) functional testing detected more control faults than did the other methods.

In comparing these results to related studies, we find mixed conclusions. A prototype analysis done at the University of Maryland in the Fall of 1981 [Hwang 81] supported the belief that code reading by stepwise abstraction does

as well as the computer-based methods, with each strategy having its own advantages. In the Myers experiment [Myers 78], the three techniques compared (functional testing, 3-person code reviews, control group) were equally effective. He also calculated that code reviews were less cost-effective than the computer-based testing approaches. The first observation is supported in one study phase here, but the other observation is not. A study conducted by Hetzel [Hetzel 76] compared functional testing, code reading, and "selective" testing (a composite of functional, structural, and reading techniques). He observed that functional and "selective" testing were equally effective, with code reading being inferior. As noted earlier, this is not supported by this analysis. The study described in this analysis examined the technique of code reading by stepwise abstraction, while both the Myers and Hetzel studies examined alternate approaches to offline (nonexecution-based) review/reading.

A few remarks are appropriate about the comparison of the cost-effectiveness and phase-availability of these testing techniques. When examining the effort associated with a technique, both fault detection and fault isolation costs should be compared. The code readers have both detected and isolated a fault; they located it in the source code. Thus, the reading process condenses fault detection and isolation into one activity. Functional and structural testers have only detected a fault; they need to delve into the source code and expend additional effort in order to isolate the defect. Also, a nonexecution-based reading process can be applied to any document produced during the development process (e.g., high-level design document, low-level design document,

source code document). While functional and structural execution-based techniques may only be applied to documents that are executable (e.g., source code), which are usually available later in the development process.

Investigations related to this work include studies of fault classification [Weiss & Basili 85, Johnson, Draper & Soloway 83, Ostrand & Weyuker 83, Basili & Perricone 84] and Cleanroom software development [Selby, Basili & Baker 85]. In the Cleanroom software development approach, techniques such as code reading are used in the development of software completely off-line (i.e., without program execution). In the above study, systems developed using Cleanroom met system requirements more completely and had a higher percentage of successful operational test cases than did systems developed with a more traditional approach.

This empirical study is intended to advance the understanding of how various software testing strategies contribute to the software development process and to one another. The results given were calculated from a set of individuals applying the three techniques to unit-sized programs – the direct extrapolation of the findings to other testing environments is not implied. However, valuable insights into software testing have been gained.

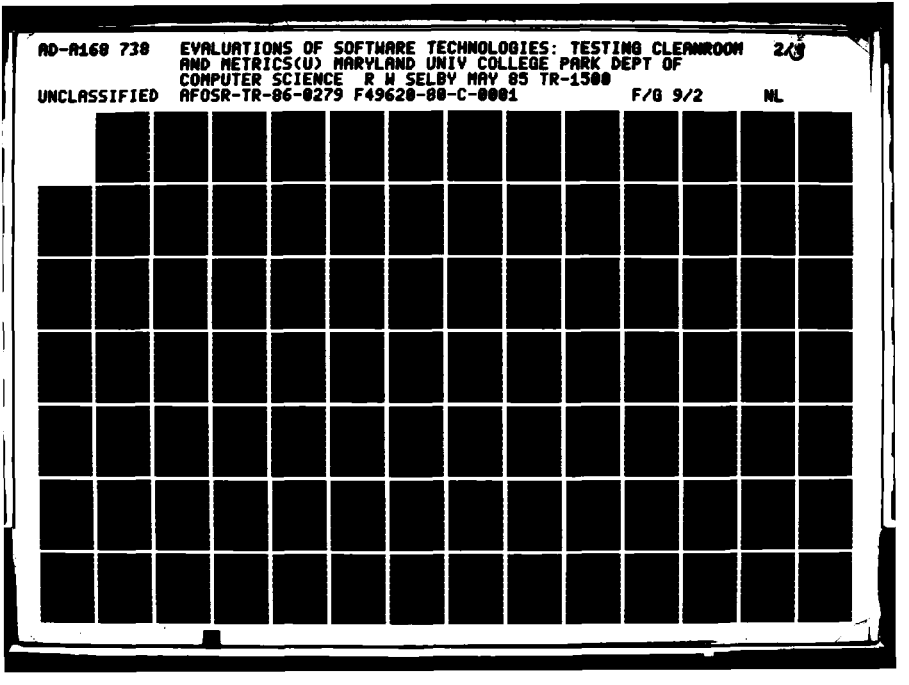
4.2. Cleanroom Development Approach Analysis

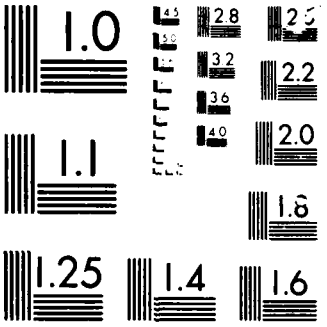
The need for discipline in the software development process and for high quality software motivates the Cleanroom software development approach. In addition to improving the control during development, this approach is intended to deliver a product that meets several quality aspects: a system that conforms with the requirements, a system with high operational reliability, and source code that is easily readable and modifiable.

The next section describes the Cleanroom approach and a framework of goals for characterizing its effects. The following section presents an empirical study using the approach. The results are then given of an analysis comparing projects developed using Cleanroom with those of a control group. The overall conclusions are presented in a final section.

4.2.1. Cleanroom Software Development Method

The Federal Systems Division of IBM [Dyer 82c, Dyer & Mills 82] presents the Cleanroom software development method as a technical and organizational approach to developing software with certifiable reliability. The idea is to deny the entry of defects during the development of software, hence the term "Cleanroom." The focus of the method is imposing discipline on the development process by integrating formal methods for specification and design, complete off-line development, and statistically based testing. These components are intended to contribute to a software product that has a high probability of zero defects and consequently a high measure of operational reliability.


AD-A168 738 EVALUATIONS OF SOFTWARE TECHNOLOGIES: TESTING CLEANROOM AND METRICS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE R W SELBY MAY 85 TR-1500
2/3
UNCLASSIFIED AFOSR-TR-86-0279 F49620-80-C-0001 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.2
2.0
1.8

The mathematically-based design methodology of Cleanroom includes the use of structured specifications and state machine models [Ferrentino & Mills 77]. A systems engineer introduces the structured specifications to restate the system requirements precisely and organize the complex problems into manageable parts [Parnas 72b]. The specifications determine the "system architecture" of the interconnections and groupings of capabilities to which state machine design practices can be applied. System implementation and test data formulation can then proceed from the structured specifications independently.

The right-the-first-time programming methods used in Cleanroom are the ideas of functionally based programming in [Mills 72a, Linger, Mills & Witt 79]. The testing process is completely separated from the development process by not allowing the developers to test and debug their programs. The developers focus on the techniques of code inspections [Fagan 76], group walkthroughs [Myers 76], and formal verification [Hoare 69, Linger, Mills & Witt 79, Shankar 82, Dyer 83] to assert the correctness of their implementation. These constructive techniques apply throughout all phases of development, and condense the activities of defect detection and isolation into one operation. This discipline is imposed with the intention that correctness is "designed" into the software, not "tested" in. The notion that "Well, the software should always be tested to find the faults" is eliminated.

In the statistically based testing strategy of Cleanroom, independent testers simulate the operational environment of the system with random testing. This testing process includes defining the frequency distribution of inputs to the sys-

tem, the frequency distribution of different system states, and the expanding hierarchy of developed system capabilities. Test cases then are chosen randomly and presented to the series of product releases, while concentrating on functions most recently delivered and maintaining the overall composite distribution of inputs. The independent testers then record observed failures and determine an objective measure of product reliability. It is believed that the prior knowledge that a system will be evaluated by random testing will affect system reliability by enforcing a new discipline into the system developers.

4.2.1.1. Investigation Goals

Some intriguing aspects of the Cleanroom approach include 1) development without testing and debugging of programs, 2) independent program testing for quality assurance (rather than to find faults or to prove "correctness" [Howden 76]), and 3) certification of system reliability before product delivery. In order to understand the effects of using Cleanroom, the following three goals are proposed: 1) characterize the effect of Cleanroom on the delivered product, 2) characterize the effect of Cleanroom on the software development process, and 3) characterize the effect of Cleanroom on the developers. An application of the goal/question/metric paradigm [Basili & Selby 84, Basili & Weiss 84] leads to the framework of goals and questions for this study appearing in Figure 20. The empirical study executed to pursue these goals is described in the following section.

Figure 20. Framework of goals and questions for Cleanroom development approach analysis.

I. Characterize the effect of Cleanroom on the delivered product.
  A. For intermediate and novice programmers building a small system, what were the operational properties of the product?
    1. Did the product meet the system requirements?
    2. How did the operational testing results compare with those of a control group?
  B. What were the static properties of the product?
    1. Were the size properties of the product any different from what would be observed in a traditional development?
    2. Were the readability properties of the product any different?
    3. Was the control complexity any different?
    4. Was the data usage any different?
    5. Was the implementation language used any differently?
  C. What contribution did programmer background have on the final product quality?

II. Characterize the effect of Cleanroom on the software development process.
  A. For intermediate and novice programmers building a small system, what techniques were used to prepare the developing system for testing submissions?
  B. What role did the computer play in development?
  C. Did they meet their delivery schedule?

III. Characterize the effect of Cleanroom on the developers.
  A. When intermediate and novice programmers built a small system, did the developers miss the satisfaction of executing their own programs?
    1. Did the missing of program execution have any relationship to programmer background or to aspects of the delivered product?
  B. How was the design and coding style of the developers affected by not being able to test and debug?
  C. Would they use Cleanroom again?

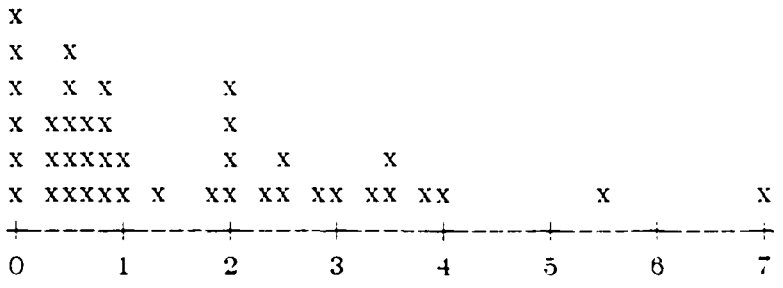
4.2.2. Empirical Study Using Cleanroom

This section describes an empirical study comparing team projects developed using Cleanroom with those using a more conventional approach.

4.2.2.1. Case Study Description

Subjects for the empirical study came from the "Software Design and Development" course taught by F. T. Baker and V. R. Basili at the University of Maryland in the Falls of 1982 and 1983. The initial segment of the course was devoted to the presentation of several software development methodologies, including top-down design, modular specification and design, PDL, chief programmer teams, program correctness, code reading, walkthroughs, and functional and structural testing strategies. For the latter part of the course, the individuals were divided into three-person chief programmer teams for a group project [Baker 72b, Mills 72b, Baker 81]. We attempted to divide the teams equally according to professional experience, academic performance, and implementation language experience. The subjects had an average of 1.8 years professional experience and were computer science majors with junior, senior, or graduate standing. Figure 21 displays the distribution of the subjects' professional experience.

Figure 21. Subjects' professional experience in years.



A requirements document for an electronic message system (read, send, mailing lists, authorized capabilities, etc.) was distributed to each of the teams. The project was to be completed in six weeks and was expected to be about 1200 lines of Simpl-T source [Basili & Turner 76]. [7] The development machine was a Univac 1100/82 running EXEC VIII, with 1200 baud interactive and remote access available.

The ten teams in the Fall 1982 course applied the Cleanroom software development approach, while the five teams in the Fall 1983 course served as a control group (non-Cleanroom). All other aspects of the developments were the same. The two groups of teams were not statistically different in terms of professional experience, academic performance, or implementation language experience. If there were any bias between the two times the course was taught, it would be in favor of the 1983 (non-Cleanroom) group because the modular design portion of the course was presented earlier. It was also the second time F. T. Baker had taught the course. Note that the teams in the non-Cleanroom group applied a development approach similar to the "disciplined team" approach examined in an earlier study [Basili & Reiter 81].

The first document every team in either group turned in contained a system specification, composite design diagram, and implementation plan. The

[7] Simpl-T is a structured language that supports several string and file handling primitives, in addition to the usual control flow constructs available, for example, in Pascal. If Pascal or FORTRAN had been chosen, it would have been very likely that some individuals would have had extensive experience with the language, and this would have biased the comparison. Also, restricting access to a compiler that produced executable code would have been very difficult.

latter element was a series of milestones describing when the various functions within the system would be available. At these various dates (minimum one week apart, maximum two), teams from both groups would then submit their systems for testing. An independent party would then apply statistically based testing to each of these deliveries and report to the team members both the successful and unsuccessful test cases. The latter would be included in the next test session for verification. Recall that the Cleanroom teams could not execute their programs – they had editing and syntax-checking capabilities only. They had to rely on the techniques of code reading, structured walkthroughs, and inspections to prepare their programs before submission. On the other hand, the non-Cleanroom teams had full access to compilation and execution facilities to test their systems prior to independent testing.

All team projects were evaluated on the use of the development techniques presented in class, the independent testing results, and a final oral interview. In addition to these sources, information on the team projects was collected from a background questionnaire, a postdevelopment attitude survey, static source code analysis, and operating system statistics. The following section briefly describes the operationally based testing process applied to all projects by the independent tester.

4.2.2.2. Operational Testing of Projects

The testing approach used in Cleanroom is to simulate the developing system's environment by randomly selecting test data from an "operational

profile," a frequency distribution of inputs to the system [Thayer, Lipow & Nelson 78, Duran & Ntafos 81]. The projects from both groups were tested interactively at the milestones chosen by each team by an independent party (i.e., R. W. Selby). A distribution of inputs to the system was obtained by identifying the logical functions in the system and assigning each a frequency. This frequency assignment was accomplished by polling eleven well-seasoned users of the University of Maryland Vax 11/780 mailing system. Then test data were generated randomly from this profile and presented to the system. Recording of failure severity and times between failure took place during the testing process. The operational statistics referred to later were calculated from fifty user-session test cases run on the final system release of each team. For a complete explanation of the operationally based testing process applied to the projects, including test data selection, testing procedure, and failure observation, see Appendix C.

4.2.3. Data Analysis and Interpretation

The analysis and interpretation of the data collected from the study appear in the following sections, organized by the goal areas outlined earlier. In order to address the various questions posed under each of the goals, some raw data usually will be presented and then interpreted. Figure 22 presents the number of source lines, executable statements, and procedures and functions to give a rough view of the systems developed.

Figure 22. System statistics.

Team	Cleanroom	Source Lines	Executable Statments	Procedures & Functions
A	yes	1681	813	55
B	yes	1626	717	42
C	yes	1118	573	42
D	yes	1046	477	30
E	yes	1087	624	32
F	yes	1213	440	35
G	yes	1196	581	31
H	yes	1876	550	51
I	yes	1305	608	23
J	yes	1052	658	24
a	no	824	410	26
b	no	1429	633	18
c	no	2264	999	46
d	no	1629	628	67
e	no	1310	459	43

4.2.3.1. Characterization of the Effect on the Product Developed

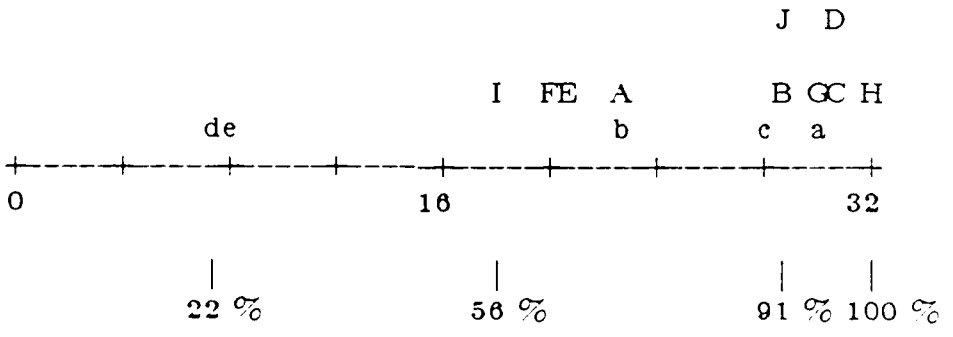
This section characterizes the differences between the products delivered by both of the development groups. Initially we examine some operational properties of the products, followed by a comparison of some of their static properties.

4.2.3.1.1. Operational System Properties

In order to contrast the operational properties of the systems delivered by the two groups, both completeness of implementation and operational testing results were examined. A measure of implementation completeness was calculated by partitioning the required system into sixteen logical functions (e.g., send mail to an individual, read a piece of mail, respond, add yourself to a mailing list, ...). Each function in an implementation was then assigned a value of

two if it completely met its requirements, a value of one if it partially met them, or zero if it was inoperable. The total for each system was calculated; a maximum score of 32 was possible. Figure 23 displays this subjective measure of requirement conformance for the systems. Note that in all figures presented, the ten teams using Cleanroom are in upper case and the five teams using a more conventional approach are in lower case. A first observation is that six of the ten Cleanroom teams built very close to the entire system. While not all of the Cleanroom teams performed equally well, a majority of them applied the approach effectively enough to develop nearly the whole product. More importantly, the Cleanroom teams met the requirements of the system more completely than did the non-Cleanroom teams.

Figure 23. Requirement conformance of the systems.



Mann-Whitney [8] signif. = .088

To compare testing results among the systems developed in the two groups, fifty random user-session test cases were executed on the final release of each system to simulate its operational environment. If the final release of a system

[8] The significance levels for the Mann-Whitney statistics reported are the probability of Type I error in an one-tailed test.

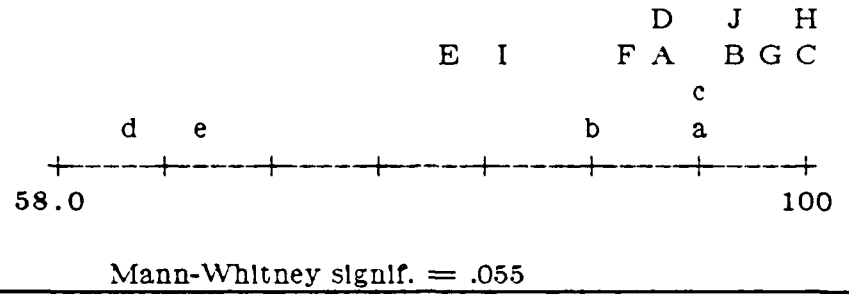
performed to expectations on a test case, the outcome was called a "success;" if not, the outcome was a "failure." If the outcome was a "failure" but the same failure was observed on an earlier test case run on the final release, the outcome was termed a "duplicate failure." Figure 24 shows the percentage of successful test cases when duplicate failures are not included. The figure displays that Cleanroom projects had a higher percentage of successful test cases at system delivery. [9] When duplicate failures are included, however, the better performance of the Cleanroom systems is not nearly as significant (MW = .134). [10] This is caused by the Cleanroom projects having a relatively higher proportion of duplicate failures, even though they did better overall. This demonstrates that while reviewing the code, the Cleanroom developers focused less than the other groups on certain parts of the system. The more uniform review of the whole system makes the performance of the system less sensitive to its operational profile. Note that operational environments of systems are usually difficult to define a priori and are subject to change.

---

[9] Although not considered here, various software reliability models have been proposed to forecast system reliability based on failure data [Musa 75, Currit 83, Goel 83].

[10] To be more succinct, MW will sometimes be used to abbreviate the significance level of the Mann-Whitney statistic.

Figure 24. Percentage of successful test cases during operational testing (without duplicate failures).



Mann-Whitney signif. = .055

In both of the product quality measures of implementation completeness and operational testing results, there was quite a variation in performance.[11] A wide variation may have been expected with an unfamiliar development technique, but the developers using a more traditional approach had a wider range of performance than did those using Cleanroom in both of the measures (even with twice as many Cleanroom teams). All of the above differences are magnified by recalling that the non-Cleanroom teams did not develop their systems in one monolithic step, they (also) had the benefit of periodic operational testing by independent testers. Since both groups of teams had independent testing of all their deliveries, the early testing of deliveries must have revealed most faults overlooked by the Cleanroom developers.

[11] An alternate perspective includes only the more successful projects from each group in the comparison of operational product quality. When the best 60% from each approach are examined (i.e., removing teams 'd,' 'e,' 'A,' 'E,' 'F,' and 'I'), the Mann-Whitney significance level for comparing implementation completeness becomes .045 and the significance level for comparing successful test cases (without duplicate failures) becomes .034. Thus, comparing the best teams from each approach increases the evidence in favor of Cleanroom in both of these product quality measures.

These comparisons suggest that the non-Cleanroom developers focused on a "perspective of the tester," sometimes leaving out classes of functions and causing a less completely implemented product and more (especially unique) failures. Off-line review techniques, however, are more general and their use contributed to more complete requirement conformance and fewer failures in the Cleanroom products. In addition to examining the operational properties of the product, various static properties were compared.

4.2.3.1.2. Static System Properties

The first question in this goal area concerns the size of the final systems. Figure 22 showed the number of source lines, executable statements, and procedures and functions for the various systems. The projects from the two groups were not statistically different (MW $>$ .10) in any of these three size attributes. Another question in this goal area concerns the readability of the delivered source code. Two aspects of reading and modifying code are the number of comments present and the density of the "complexity." In an attempt to capture the complexity density, syntactic complexity [Basili & Hutchens 83] was calculated and normalized by the number of executable statements. In addition to control complexity, the syntactic complexity metric considers nesting depth and prime program decomposition [Linger, Mills & Witt 79]. The developers using Cleanroom wrote code that was more highly commented (MW = .089) and had a lower complexity density (MW = .079) than did those using the traditional approach. A calculation of either software sci-

ence effort [Halstead 77], cyclomatic complexity [McCabe 76], or syntactic complexity without any size normalization, however, produced no significant differences (MW > .10). This seems as expected because all the systems were built to meet the same requirements.

Comparing the data usage in the systems, Cleanroom developers used a greater number of global data items (MW = .071). Also, Cleanroom projects possessed a higher percentage of assignment statements (MW = .056). These last two observations could be a manifestation of teaching the Cleanroom subjects modular design later in the course (see Case Study Description), or possibly an indication of using the approach.

Some interesting observations surface when the operational quality measures of the Cleanroom products are correlated with the usage of the implementation language. Both percentage of successful test cases (without duplicate failures) and implementation completeness correlated with percentage of procedure calls (Spearman R = .65, signif. = .044, and R = .57, signif. = .08, respectively) and with percentage of if statements (R = .62, signif. = .058, and R = .55, signif. = .10, respectively). However, both of these two product quality measures correlated negatively with percentage of case statements (R = -.86, signif. = .001, and R = -.69, signif. = .027, respectively) and with percentage of while statements (R = -.65, signif. = .044, and R = -.49, signif. = .15, respectively). There were also some negative correlations between the product quality measures and the average software science effort per subroutine (R = -.52, signif. = .12, and R = -.74, signif. = .013, respectively) and the average

number of occurrences of a variable (R = –.54, signif. = .11, and R = –.56, signif. = .09, respectively). Considering the products from all teams, both percentage of successful test cases (without duplicate failures) and implementation completeness had some correlation with percentage of if statements (R = .48, signif. = .07, and R = .45, signif. = .09, respectively) and some negative correlation with percentage of case statements (R = –.48, signif. = .07, and R = –.42, signif. = .12, respectively). Neither of the operational product quality measures correlated with percentage of assignment statements when either all products or just Cleanroom products were considered. These observations suggest that the more successful Cleanroom developers simplified their use of the implementation language; i.e., they used more procedure calls and if statements, used fewer case and while statements, had a lower frequency of variable reuse, and wrote subroutines requiring less software science effort to comprehend.

4.2.3.1.3. Contribution of Programmer Background

When examining the contribution of the Cleanroom programmers' background to the quality of their final products, general programming language experience correlated with percentage of successful operational tests (without duplicate failures: Spearman R = .66, signif. = .04; with duplicates: R = .70, signif. = .03) and with implementation completeness (R = .55; signif. = .10). No relationship appears between either operational testing results or implementation completeness and either professional[12] or testing experience. These

[12] In fact, there are very slight negative correlations between years of professional experience and both percentage of successful tests (without duplicate

background/quality relations seem consistent with other studies [Curtis 83].

4.2.3.1.4. Summary of the Effect on the Product Developed

In summary, Cleanroom developers delivered a product that 1) met system requirements more completely, 2) had a higher percentage of successful test cases, 3) had more comments and less dense complexity, and 4) used more global data items and a higher percentage of assignment statements. The more successful Cleanroom developers 1) used more procedure calls and if statements, 2) used fewer case and while statements, 3) reused variables less frequently, 4) developed subroutines requiring less (software science) effort to comprehend, and 5) had more general programming language experience.

4.2.3.2. Characterization of the Effect on the Development Process

In a postdevelopment attitude survey, the developers were asked how effectively they felt they applied off-line review techniques in testing their projects (see Figure 25). This was an attempt to capture some of the information necessary to answer the first question under this goal (question II.A). In order to make comparisons at the team level, the responses from the members of a team are composed into an average for the team. The responses to the question appear on a team basis in a histogram in the second part of the figure. Of the Cleanroom developers, teams 'A,' 'D,' 'E,' 'F,' and 'I' were the least confident in their use of the off-line review techniques and these teams also performed the

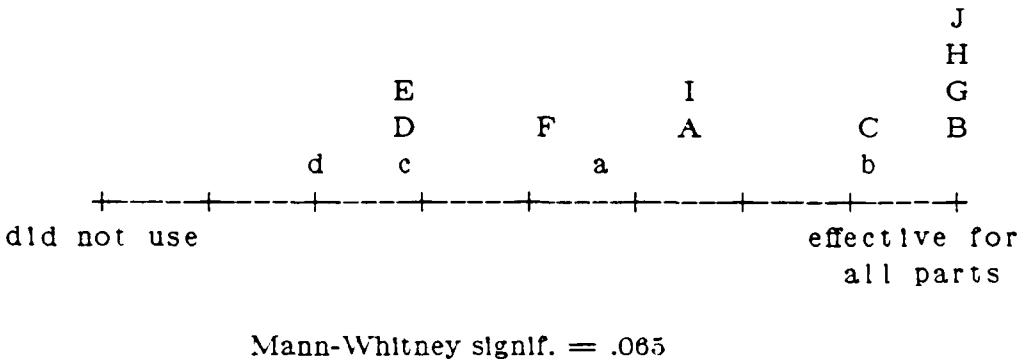
---

failures: R = -.46, signif. = .18) and implementation completeness (R = - .47, signif. = .17).

worst in terms of operational testing results; four of these five teams performed the worst in terms of implementation completeness. Off-line review effectiveness correlated with percentage of successful operational tests (without duplicate failures) for the Cleanroom teams (Spearman R = .74; signif. = .014) and for all the teams (R = .76; signif. = .001); it correlated with implementation completeness for all the teams (R = .58; signif. = .023). Neither professional nor testing experience correlated with off-line review effectiveness when either all teams or just Cleanroom teams were considered.

Figure 25. Breakdown of responses to the attitude survey question, "Did you feel that you and your team members effectively used off-line review techniques in testing your project?". (Responses are from Cleanroom teams.) [13]

14 – Yes, they were effective for testing all parts of the program
5.5 – We used them but felt that they were only appropriate for certain parts of the program
8.5 – We used them occasionally, but they were not really a major contributing factor to the development
0 – Did not really use them at all



Mann-Whitney signif. = .065

The histogram in Figure 25 shows that the Cleanroom developers felt they applied the off-line review techniques more effectively than did the non-Cleanroom teams. The non-Cleanroom developers were asked to give a relative breakdown of the amount of time spent applying testing and verification techniques. Their aggregate response was 39% off-line review, 52% functional testing, and 9% structural testing. From this breakdown, we observe that the

[13] There are half-responses because an individual checked both the second and third choices. The responses total to 28, not 30, because two separate teams lost a member late in the project. (See Distinction Among Teams).

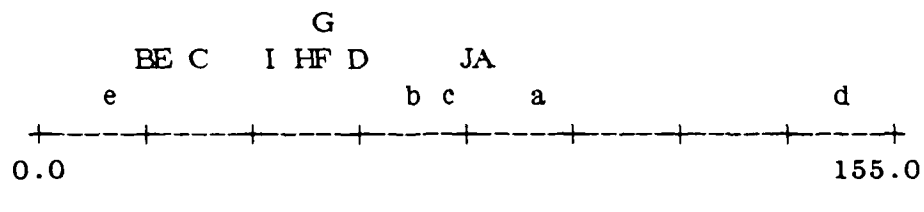
non-Cleanroom teams primarily relied on functional testing to prepare their systems for independent testing. Since the Cleanroom teams were unable to rely on testing methods, they may have (felt they had) applied the off-line review techniques more effectively.

Since the role of the computer is more controlled when using Cleanroom, one would expect a difference in on-line activity between the two groups. Figure 26 displays the amount of connect time that each of the teams cumulatively used. A comparison of the cpu-time used by the teams was less statistically significant (MW = .110). Neither of these measures of on-line activity related to how effectively a team felt they had used the off-line techniques when either all teams or just Cleanroom teams were considered. Although non-Cleanroom team 'd' did a lot of on-line testing and non-Cleanroom team 'e' did little, both teams performed poorly in the measures of operational product quality discussed earlier. The operating system of the development machine captured these system usage statistics. Note that the time the independent party spent testing is included. [14] These observations exhibit that Cleanroom developers spent less time on-line and used fewer computer resources. These results empirically support the reduced role of the computer in Cleanroom development.

---

[14] When the time the independent tester spent is not included, the significance levels for the non-parametric statistics do not change.

Figure 26. Connect time in hours during project development. [15]

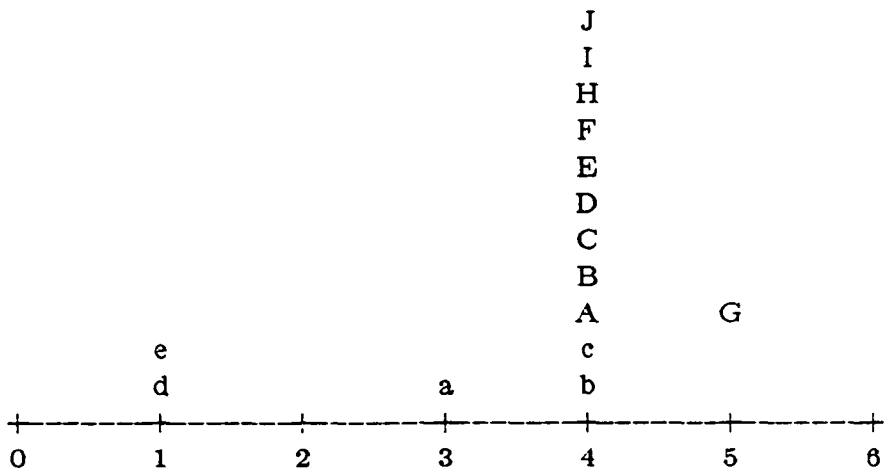


Mann-Whitney signif. = .089

Schedule slippage continues to be a problem in software development. It would be interesting to see whether the Cleanroom teams demonstrated any more discipline by maintaining their original schedules. All of the teams from both groups planned four releases of their evolving system, except for team 'G' which planned five. Recall that at each delivery an independent party would operationally test the functions currently available in the system, according to the team's implementation plan. In Figure 27, we observe that all the teams using Cleanroom kept to their original schedules by making all planned deliveries; only two non-Cleanroom teams made all their scheduled deliveries.

[15] Non-Cleanroom team 'e' entered a substantial portion of its system on a remote machine, only using the Univac computer mainly for compilation and execution. (See Distinction Among Teams.)

Figure 27. Number of system releases.



Mann-Whitney signif. = .006

4.2.3.2.1. Summary of the Effect on the Development Process

Summarizing the effect on the development process, Cleanroom developers 1) felt they applied off-line review techniques more effectively, while non-Cleanroom teams focused on functional testing; 2) spent less time on-line and used fewer computer resources; and 3) made all their scheduled deliveries.

4.2.3.3. Characterization of the Effect on the Developers

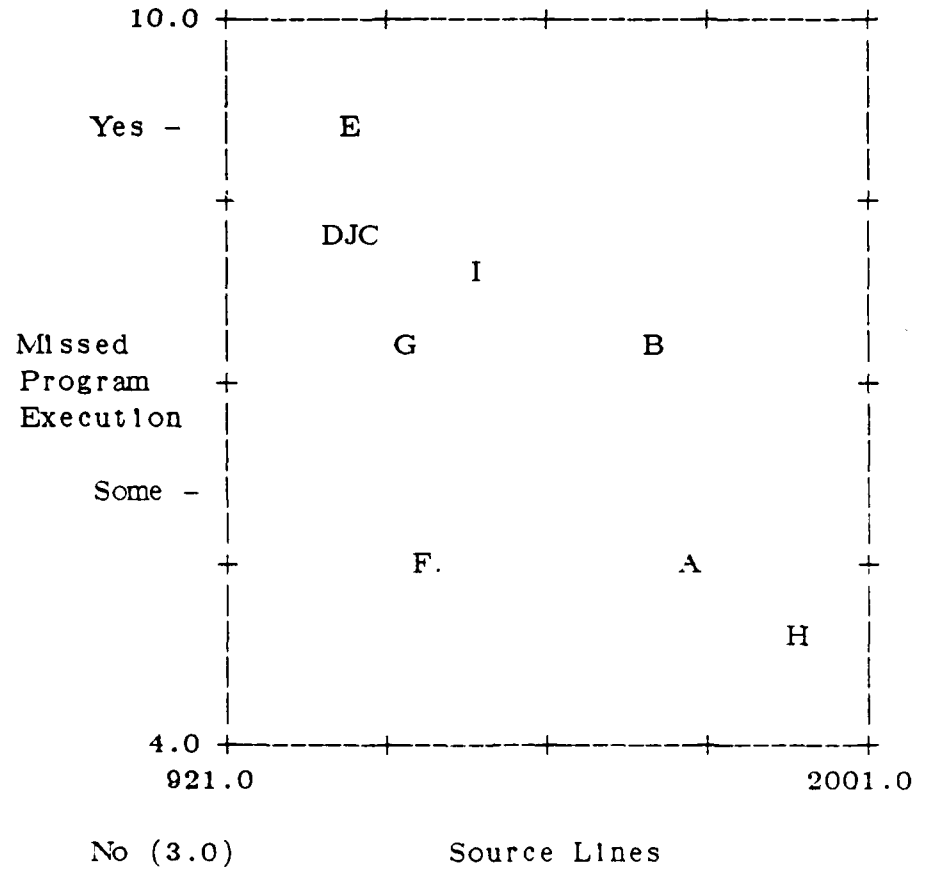
The first question posed in this goal area is whether the individuals using Cleanroom missed the satisfaction of executing their own programs. Figure 28 presents the responses to a question included in the postdevelopment attitude survey on this issue. As might be expected, almost all the individuals missed some aspect of program execution. As might not be expected, however, this missing of program execution had no relation to either the product quality

measures mentioned earlier or the teams' professional or testing experience. Also, missing program execution did not increase with respect to program size (see Figure 29).

Figure 28. Breakdown of responses to the attitude survey question, "Did you miss the satisfaction of executing your own programs?".

13 – Yes, I missed the satisfaction of program execution.
11 – I somewhat missed the satisfaction of program execution.
4 – No, I did not miss the satisfaction of program execution.

Figure 29. Relationship of program size vs. missing program execution.



Spearman correlations: –.85 (signif. = .002) with source lines; –.70 (signif. = .03) with number separately compilable modules; –.57 (signif. = .09) with number procedures and functions.

Figure 30 displays the replies of the developers when they were asked how their design and coding style was affected by not being able to test and debug. At first it would seem surprising that more people did not modify their develop-

ment style when applying the techniques of Cleanroom. Several persons mentioned, however, that they already utilized some of the ideas in Cleanroom. Keeping a simple design supports readability of the product and facilitates the processes of modification and verification. Although some of the objective product measures presented earlier showed differences in development style, these subjective ones are interesting and lend insight into actual programmer behavior.

Figure 30.
Breakdown of responses to the attitude survey question, "How was your design and coding style affected by not being able to test and debug?".

2 – Yes, my style was substantially revised. 15 – I modified some of my tendencies. 11 – It did not affect my style at all.
Frequently mentioned responses include – kept design simple, attempted nothing fancy – kept readability of code in mind – already was a user of off-line review techniques – very careful scrutiny of code for potential mistakes – prepared for a larger range of inputs

One indicator of the impression that something new leaves on people is whether they would do it again. Figure 31 presents the responses of the individuals when they were asked whether they would choose to use Cleanroom as either a software development manager or as a programmer. Even though these responses were gathered (immediately) after course completion, subjects desiring to "please the instructor" may have responded favorably to this type of question regardless of their true feelings. Practically everyone indicated a willingness to apply the approach again. It is interesting to note that a greater

number of persons in a managerial role would choose to always use it. Of the persons that ranked the reuse of Cleanroom fairly low in each category, four of the five were the same people. Of the six people that ranked reuse low, four were from less successful projects (one from team 'A', one from team 'E' and two from team 'I'), but the other two came from reasonably successful developments (one from team 'C' and one from team 'J'). The particular individuals on teams 'E,' 'I,' and 'J' rated the reuse fairly low in both categories.

Figure 31.
Breakdown of responses to the attitude survey question, "Would you use Cleanroom again?". (One person did not respond to this question.)

As a software development manager?
- 8 – Yes, at all times
- 14 – Yes, but only for certain projects
- 5 – Not at all

As a programmer?
- 4 – Yes, for all projects
- 18 – Yes, but not all the time
- 5 – Only if I had to
- 0 – I would leave if I had to

4.2.3.3.1. Summary of the Effect on the Developers

In summary of the effect on the developers, most Cleanroom developers 1) modified in part their development style, 2) missed program execution, and 3) indicated they would use the approach again.

4.2.3.4. Distinction Among Teams

In spite of efforts to balance the teams according to various factors (see Case Study Description), a few differences among the teams were apparent.

Two separate Cleanroom teams, 'H' and 'I,' each lost a member late in the project. Thus at project completion, there were eight three-person and two two-person Cleanroom teams. Recall that team 'H' performed quite well according to requirement conformance and testing results, while team 'I' did poorly. Also, the second group of subjects did not divide evenly into three-person teams. Since one of those individuals had extensive professional experience, non-Cleanroom team 'e' consisted of that one highly experienced person. Thus at project completion, there were four three-person and one one-person non-Cleanroom teams. Although team 'e' wrote over 1300 source lines, this highly experienced person did not do as well as the other teams in some respects. This is consistent with another study in which teams applying a "disciplined methodology" in development outperformed individuals [Basili & Reiter 81]. Figure 32 contains the significance levels for the above results when team 'e,' when teams 'H' and 'I,' and when teams 'e,' 'H,' and 'I' are removed from the analysis. Removing teams 'H' and 'I' has little effect on the significance levels, while the removal of team 'e' causes a decrease in all of the significance levels except for executable statements, software science effort, cyclomatic complexity, syntactic complexity, connect-time, and cpu-time.

Figure 32. Summary of measure averages and significance levels.

Measure	Average		Mann-Whitney significance levels			
	Clean-room Teams	Non-Clean-room Teams	All Teams	Without Team e	Without Teams H.I	Without Teams e.H.I
Source lines	1320.0	1491.2	.196	.240	.153	198
Executable stmts	604.1	625.4	.500	.286	.442	.367
#Procedures & functions	36.5	40.0	.357	.500	.330	.500
%Implementation completeness	82.5	60.0	.088	.197	.093	.196
%Successful tests (w/o duplicate failures)	92.5	80.8	.055	.128	.053	.116
%Successful tests (w/ duplicate failures)	78.7	59.2	.134	.285	.151	.304
#Comments	194.9	122.2	.089	.102	.190	198
Syntactic complexity/ executable stmts	1.5	1.6	.079	.179	.082	.175
Software Science E	6728.6e3	7355.4e3	.451	.240	.442	248
Cyclomatic complexity	196.8	212.2	.250	.198	.255	248
Syntactic complexity	917.5	1017.0	.500	286	.500	.305
#Global data items	37.6	24.2	.071	.129	.053	.117
%Assignment stmts	34.2	26.6	.056	.129	.040	.087
Off-line effectiveness	3.2	2.5	.065	065	.098	.098
Connect-time (hr.)	41.0	71.3	.089	.012	.121	021
Cpu-time (min.)	71.7	136.1	.110	.017	.072	.009
#Deliveries	4.1	2.6	.006	.015	.010	022

4.2.4. Conclusions

This paper describes "Cleanroom" software development - an approach intended to produce highly reliable software by integrating formal methods for specification and design, complete off-line development, and statistically based testing. The goal structure, experimental approach, data analysis, and conclusions are presented for a replicated-project study examining the Cleanroom approach. This is the first investigation known to the authors that applied Cleanroom and characterized its effect relative to a more traditional development ap-

proach.

The data analysis presented and the testimony provided by the developers suggest that the major results of this study are 1) most developers were able to apply the techniques of Cleanroom effectively; 2) the Cleanroom teams' products met system requirements more completely and had a higher percentage of successful test cases; 3) the source code developed using Cleanroom had more comments and less dense complexity; 4) the use of Cleanroom successfully modified aspects of development style; and 5) most Cleanroom developers indicated they would use the approach again.

It seems that the ideas in Cleanroom help attain the goals of producing high quality software and increasing the discipline in the software development process. The complete separation of development from testing appears to cause a modification in the developers' behavior, resulting in increased process control and in more effective use of formal methods for software specification, design, off-line review, and verification. It seems that system modification and maintenance would be more easily done on a product developed in the Cleanroom method, because of the product's thoroughly conceived design and higher readability. Thus, achieving high requirement conformance and high operational reliability coupled with low maintenance costs would help reduce overall costs, satisfy the user community, and support a long product lifetime.

This empirical study is intended to advance the understanding of the relationship between introducing discipline into the development process (as in Cleanroom) and several aspects of product quality: conformance with require-

ments, high operational reliability, and easily modifiable source code. The results given were calculated from a set of teams applying Cleanroom development on a relatively small project – the direct extrapolation of the findings to other projects and development environments is not implied. Valuable insights, however, have been gained from the analysis.

4.3. Characteristic Metric Set Study

Several metrics have been proposed to predict product cost/quality and to capture distinct project aspects [McCabe 76, Halstead 77, Chen 78, Gaffney & Heller 80, Behrens 83]. The effectiveness of the measures in capturing what is intended, however, has depended on the particular environment examined [Walston & Felix 77, Curtis, Sheppard & Milliman 79, Feuer & Fowlkes 79, Basili 80, Bailey & Basili 81, Boehm 81, Brooks 81, Zolnowski & Simmons 81, Vosburgh et al. 84]. A particular software metric that has been useful to characterize, evaluate, or predict aspects of software development in one environment may have limited usefulness elsewhere. The differing cost/quality goals among environments and the diversity in methodology, software type, etc. contribute to the inconsistent performance of metrics. Thus, it is inappropriate to attempt to select a set of software metrics that have universal effectiveness across all software environments. The selection of a set of metrics appropriate for a particular environment must consider its individual features; that is, a metric set must be customized to a specific environment.

This study develops an approach for customizing to an environment a characteristic set of cost and quality measures. The approach then is applied in a software production environment. This section describes the concept of a characteristic software metric set, investigation goals, empirical study, and data analysis.

4.3.1. Characteristic Software Metric Sets

The successful management of software projects requires a diverse range of capabilities, including monitoring and controlling the evolving software system and forecasting the outcome of the development. Techniques that assist in these management functions may lead to more successful projects, and possibly higher product requirement conformance and operational reliability. The idea of a characteristic software metric set supports several aspects of software management.

A characteristic software metric set is a concise collection of measures that capture distinct factors in a software development/modification environment. A characteristic metric set can be thought of as a vector of measures that represents different areas of importance in an environment. Since both cost/quality goals and production environments differ, the particular factors that are captured by the metrics in the set will differ across environments. The calculation of a characteristic metric set should be based on the particular cost and quality goals in an environment, and reflect the inherent differences of the environment from others.

A characteristic metric set may be used to 1) characterize an environment, 2) compare an environment with others, 3) monitor current project status, or 4) forecast project outcome relative to past projects, when metrics in the set are available early in development. Once the distinct factors in an environment's set are determined, the set then characterizes what aspects are important in the environment. Comparing the characteristic set of factors in one environment

with the sets of other environments provides a format to distinguish and contrast among them. Within an individual environment, the actual values of the metrics in the set characterize a particular project or project subsystem. The change in the metric values during a project can be used to monitor project status and its change over time. The characteristic set in conjunction with historical data can be used to forecast the outcome of the current project relative to past project performance.

4.3.1.1. Investigation Goals

The goals for this study are threefold. I.) Develop an approach for customizing a set of measures to particular cost/quality goals in a specific environment. II.) Apply the approach to calculate the characteristic set for the NASA/SEL environment. III.) Examine the usability of the approach as a management tool for predicting outcome of system parts. An application of the goal/question/metric paradigm [Basili & Selby 84, Basili & Weiss 84] leads to the framework of goals and questions for this study appearing in Figure 33.

Figure 33. Framework of goals and questions for characteristic set study.

I. Develop an approach for customizing a set of measures to particular cost/quality goals in a particular environment.

   A. Is the approach sensitive to different cost and quality goals?

   B. Does the approach capture the aspects that distinguish a given environment from others?

II. Calculate the characteristic set for the NASA/SEL environment.

A. In the NASA/SEL environment of projects and programmers, which distinct factors are important?

1. What is the ordering of factors that reflects their importance in the environment?

2. How many distinct factors are there?

B. What metrics are appropriate for the various factors in the set?

III. Examine the usability of the approach as a management tool for predicting outcome of system parts.

A. In the NASA/SEL environment of projects and programmers, does determining a characteristic metric set and using historical data enable one to identify which modules will have interesting attributes, such as high total development effort?

B. What are the best single identifiers of interesting modules when the cost/quality aspect considered changes?

4.3.2. Empirical Study

This section describes the SEL environment examined and the scheme for data collection.

4.3.2.1. SEL Environment

The Software Engineering Laboratory (SEL) [Basili et al. 77, Basili & Zelkowitz 78, Card et al. 82, SEL 82] is a joint venture between the University of Maryland, NASA/Goddard Space Flight Center, and Computer Sciences Corporation. The purpose of the SEL has been to provide an experimental database for examining relationships among the factors that affect the software development process and the delivered product. The software comprising the database is ground support software for satellites. The six systems analyzed in

this study consisted of 51,000 to 112,000 lines of FORTRAN source code, and took between 6900 and 22,300 man-hours to develop over a period of 9 to 21 months. There are from 200 to 600 modules (e.g., subroutines) in each system and the staff size ranges from 8 to 23 people per project, including the support personnel. Anywhere from 10 to 61 percent of the source code is reused or modified from previous projects.

4.3.2.2. Effort, Change, and Fault Data

The data discussed in this study are extracted from several sources. Among the data analyzed are the effort to design, code, and test the various modules of the systems as well as the changes and faults that occurred during their development. Effort data were obtained from a collection form that is filled out weekly by all programmers on the project. They report the time they spent on each module in the system partitioned into the phases of design, code, and test, as well as any other time they spend on work related to the project, e.g., documentation, meetings, etc. A module is defined as any named object in the system; that is, a module is either a main procedure, block data, subroutine or function. The faults and changes are reported on another data collection form that is completed by a programmer each time a change is made to the system. A static code analysis program called SAP [Decker & Taylor 82] automatically computed several of the static metrics examined in this analysis.

4.3.3. Data Analysis

The following sections present the analysis and results from this study broken down by the goal areas outlined earlier.

4.3.3.1. Approach for Set Calculation

A proposed approach for calculating a characteristic set consists of three steps: 1) formulate the goals and questions that represent cost/quality factors in an environment; 2) list all measures that capture information relating to the goals; and 3) condense measures into a set capturing distinct factors. This approach satisfies the two key aspects of customizing a characteristic metric set to an environment: sensitivity to the cost/quality goals of importance in the environment, and capturing the features that give the environment its identity.

The first step is to generate a goal and question framework for the environment on which to base the generation of all potential metrics. After the goals and questions have been specified for an environment, all possible metrics are listed that represent relevant information. These first two steps are an application of the goal/ question/ metric paradigm [Basili & Weiss 84, Basili & Selby 84]. Since a software environment is in some sense defined by the projects it develops, applying the metrics listed to those projects reflects an environment's distinguishing features. The third step is to condense the collection of measures into a characteristic set. Factor analysis may be applied to accomplish this step. This data reduction task actually groups the metrics listed according to how they relate to the distinct factors in an environment. Appropriate metrics

that relate to each of the factors can then be selected based on some criteria, such as ease of calculation or phase availability.

4.3.3.1.1. An Alternate Approach

An alternate approach to determining a small set of characteristic measures was examined in [Elshoff 84]. In this approach, twenty candidate complexity measures were calculated on 585 PL/I procedures. The name of each procedure was put into a large "complexity pot" once for each time the procedure appeared in the top decile of a candidate complexity measure. Since there were twenty candidate measures, the name of a given procedure could then appear up to twenty times in the pot. The procedures identified by a single measure were then compared with those in the total pot. For each appearance of a procedure name in the total pot, a candidate measure was awarded one point if that name was in the measure's top decile. The candidate complexity measure that scored the highest would be selected for the characteristic set. All occurrences of procedure names were then removed from the pot that appeared in the top decile of the first measure selected. The scores for the measures were then recalculated based on the remaining procedures, and another measure would then be selected, continuing until no procedures remained in the pot.

This alternate approach suffers because of the biased technique used to select measures in the characteristic set, and a troublesome fundamental assumption in the calculation. Including a large number of highly dependent program measures in the collection examined (e.g., the software "quantity" group

of executable statements, length, volume, vocabulary, ...) increased disproportionately the number of appearances of routines commonly selected by that group in the pot of "complex" programs. It is therefore no surprise that the measure that selected the greatest percentage of the appearances in the pot is one member of the "quantity" group (length). In each of the twenty program measures examined, the top decile of programs was chosen as the most complex according to that measure. This decision relied on the implicit assumption that software complexity is a monotonically increasing function of each of the measures, which is possibly troublesome.

The approach presented here bases the selection of a characteristic set of measures on aspects of cost and quality in an environment. The use of measures in the characteristic set to identify modules with particular attributes, such as those of high "complexity" as was done in [Elshoff 84], is discussed in "Use as a Management Tool."

4.3.3.2. Application in the SEL Environment

In the application of the approach in the NASA/SEL environment, there were two major reasons to use just six recent projects. First, changes and improvements in development technologies and personnel tend to be reflected in the projects developed (as they are intended to be). Therefore, the consideration of projects not recently completed would not be representative of the current environment. Second, several development environments do not have a long history of data collection. Discussing an approach that required a large

project database would have little utility for them.

Three goal areas were defined for the SEL environment. The first goal area was to analyze the system development effort. An example question under this goal is "What are the attributes of modules that result in high development effort?". The second goal area was to analyze the system modifications. An example question here is "What are the attributes of modules that will be difficult to change?". Analyzing the system faults was the third goal area. An example question would be "What are the attributes of modules that will be fault-prone?". The generated list of measures based on these three goal areas appears in Figure 34; a total of 65 measures was examined. The measures are grouped according to the general areas of size/complexity [McCabe 76], effort, faults/changes, and software science [Halstead 77]. The set notation in the figure signifies the normalization of one metric by another, e.g., amount of design effort was considered alone and normalized by the amount of code effort, testing effort, and overhead effort. In addition to being examined alone, the effort and faults/changes measures were in general normalized over the size/complexity measures.

Figure 34. List of measures examined in the SEL environment.
Size/Complexity Area
source lines (SRC) executable statements (XQT) comments comments/SRC XQT/(SRC-comments) Cyclomatic_complexity Cyclomatic_complexity_2 calls {Cyclomatic_complexity, Cyclomatic_complexity_2} over {SRC, XQT}
Effort Area
total_effort design_effort code_effort testing_effort {design_effort} over {code_effort, testing_effort, overhead_effort} {code_effort} over {testing_effort, overhead_effort} {testing_effort} over {overhead_effort} {design_effort, code_effort, testing_effort} over {total_effort, calls, η_2^*} {total_effort} over {SRC, SRC-comments, XQT, calls, η_2^*}
Faults/Changes Area
version total_changes weighted_changes total_faults weighted_faults {total_faults, weighted_faults} over {SRC, XQT}
Software Science
η_1 η_2 η_2^* N1 N2/η_2 N N^ V V* L L^ 1/L 1/L^ E E^ E^^ E* B^ lambda E/SRC

From the six projects, this analysis focuses on 652 newly developed modules with complete data for the measures listed in Figure 34. The use of factor analysis isolated the set of six distinct factors including, in order of overall importance, {size, effort, η_2^*, fault density, code and test effort, #changes}. The η_2^* metric is the number of I/O parameters in a module. Some

appropriate measures that related well to each of the factors in the set were a) size – source lines, executable statements, and η_1 (the number of unique operators); b) effort – design effort, η_2^*, and testing effort / η_2^*; c) $\eta_2^* - \eta_2^*$; d) fault density – #faults / executable statement; e) code and test effort – code effort, code effort / #subroutine calls; and f) #changes – number of module versions. Thus, a feasible characteristic metric set for the SEL environment is {source lines, design effort, number of I/O parameters, fault correction effort per executable statement, code effort, number of versions}.

4.3.3.3. Use as a Management Tool

Although a characteristic set has the several uses outlined earlier, this study focuses on the use of measures in the set to forecast the outcome of modules in projects. Several studies have pointed to the unsatisfactory use of metrics as direct predictors of software cost and quality [Hamer & Frewin 82; Basili, Selby & Phillips 83; Shen, Conte & Dunsmore 83]. This inadequacy motivates the use of software metrics from a new perspective – the examination of how well the metrics in the characteristic set can identify system parts (or whole systems) resulting in high or low cost/quality. System parts with interesting cost or quality attributes include those with high/low development effort, high/low modification effort, or high/low fault correction effort.

An approach for using metrics to identify system parts having interesting attributes is as follows. First, select some interesting cost or quality aspect of a system part, such as the total development effort for a module. Then, choose a

window of modules that is useful to identify, such as those modules that will be in a project's upper quartile of development effort. Next, determine the ranges of metric values that contained modules from past projects ending up in the upper quartile of development effort. From the calculation of the sensitive metric ranges and the use of conditional probabilities from historical data, this approach is intended to be able to identify interesting modules in the system.

4.3.3.3.1. Conditional Probabilities from Historical Data

The conditional probabilities displayed in Figure 35 were calculated from six SEL projects, and are interpreted as follows. The table is divided into three sections, corresponding to the three SEL goal areas discussed above. There is a table section for each dependent variable: total module development effort, total effort for module modification, and total effort for fault correction in a module. The characteristic set of six metrics that represents the different environmental factors is listed in each section of the table. Consider the section on total module development effort. The entries in the table are the probability that a module's eventual outcome will be in the upper quartile of total module development effort, given that a module is currently in quartile Q_j of metric M_i. For example, given that a module is in the upper quartile of code effort, it has a probability of .74 of ending up in the upper quartile of total module development effort. A module in the third quartile of source lines has a probability of just .14 of ending up in the upper quartile of total development effort. The interpretation is the same for the other dependent variables of module

modification effort and module fault correction effort. Figure 36 is analogous to Figure 35, except that the entries are the conditional probabilities that the eventual outcome will be in the *lower* (instead of the upper) quartile of the respective dependent variable. For example, a module in the lower quartile of number of versions has a probability of .50 of ending up in the lower quartile of total module development effort.

Figure 35. Conditional probabilities based on SEL data: **upper** quartiles of dependent variables.

	Quartile of Metric M_i				
Dependent Variable	Upper	Second	Third	Lower	Characteristic Set Metric M_i
Module Development Effort					
	.74	.18	.04	.04	code effort
	.56	.18	.13	.13	design effort
	.51	.26	.14	.09	source lines
	.48	.24	.17	.11	η_2^*
	.44	.37	.13	.06	version
	.41	.28	.15	.16	fault correction effort / XQT
Module Modification Effort					
	.65	.18	.08	.09	fault correction effort / XQT
	.52	.33	.11	.04	version
	.50	.27	.17	.06	code effort
	.50	.28	.13	.09	source lines
	.45	.24	.23	.08	η_2^*
	.41	.25	.18	.17	design effort
Module Fault Correction Effort					
	.81	.19	.00	.00	fault correction effort / XQT
	.50	.35	.12	.03	version
	.48	.29	.15	.08	code effort
	.42	.33	.14	.11	source lines
	.42	.28	.19	.11	η_2^*
	.36	.25	.20	.19	design effort

Figure 36. Conditional probabilities based on SEL data: **lower** quartiles of dependent variables.

	Quartile of Metric M_i				
Dependent Variable	Upper	Second	Third	Lower	Characteristic Set Metric M_i
Module Development Effort					
	.00	.00	.23	.77	code effort
	.10	.12	.24	.54	source lines
	.06	.14	.30	.50	version
	.09	.21	.25	.45	η_2^*
	.02	.23	.37	.38	design effort
	.12	.25	.32	.31	fault correction effort / XQT
Module Modification Effort					
	.09	.15	.28	.48	version
	.01	.13	.43	.43	fault correction effort / XQT
	.14	.19	.25	.42	η_2^*
	.11	.18	.30	.41	source lines
	.11	.18	.34	.37	code effort
	.18	.28	.27	.27	design effort
Module Fault Correction Effort					
	.00	.00	.50	.50	fault correction effort / XQT
	.18	.18	.27	.37	version
	.21	.19	.29	.31	source lines
	.18	.24	.27	.31	code effort
	.20	.24	.25	.31	η_2^*
	.18	.25	.29	.28	design effort

4.3.3.3.2. Data Interpretation

The information in these tables could be used to forecast the outcome of modules in a system. At the end of the design phase, the η_2^* metric and the

amount of effort spent in design are known. The modules in the upper quartile of design effort should be identified by a project manager because these modules have a probability of .56 of ending up in the upper quartile of total development effort. That is, in this environment the modules in the upper quartile of design effort are more than twice (=.56/.25) as likely than by chance to be the most expensive to develop overall; these modules are approximately 28 (=.56/.02) times more likely to be in the upper quartile of total development effort than to be in the lower quartile of total development effort. Modules in the upper quartile of the η_2^* metric are almost twice as likely than by chance to require the most effort to develop, modify, and correct. Other observations include 1) it is easiest to identify those modules that will have high development effort; 2) it is most *difficult to identify those* modules that will require little fault correction effort; and 3) the metrics of design effort and η_2^* are reasonably similar in forecasting ability, except that η_2^* is superior in identifying modules that will require little modification effort.

The two tables above help characterize the SEL development environment. The total development effort for a module tends to be indicated by the module's coding effort – modules in the extreme quartiles of coding effort are three times more likely than by chance to be in the corresponding extreme quartiles of total development effort. Since the programmers in the SEL are quite experienced in the application and with appropriate design approaches, the dominance of coding effort seems reasonable. In other environments, the amount of design effort might better indicate the total development effort required. Other observations

include 1) high density of fault correction effort (fault correction effort per executable statement) indicates high total modification effort and high total fault correction effort; and 2) an extreme (high or low) number of program versions reflects a corresponding amount of modification effort and of correction effort.

Ideally, the metrics in the characteristic set would all be available early in development and have strong relationships with the dependent variables of interest. Some measures, such fault correction effort per executable statement, have limited usefulness as a forecaster because of not being available until late in project development. An assumption is needed in order to use conditional probabilities from past projects to forecast the outcome of modules from a current project. The assumption is that the relationship between a module's metric value (at a point in time) and its eventual outcome is the same as the relationship between the metric values from past projects' modules (at a corresponding point in time) and their eventual outcome. When using data based on recent projects that were similar to the current one, this assumption is reasonable. Note that the examples and conditional probabilities presented are from a particular environment, project data from other environments may differ.

Using a characteristic metric set with conditional probabilities from past projects enables the monitoring of a small set of customized measures to forecast the outcome of the current project. A characteristic set is usable as a management tool as soon as the metrics in the set are available.

4.3.4. Conclusions

A characteristic software metric set is intended to help support the effective management of software development and modification. The approach examined for building a characteristic metric set is adaptable to different cost/quality goals and to different environments. The calculation and use of the set could be coupled to an automated project monitor and database. The major results of this study are 1) an approach has been described for customizing a characteristic software metric set to an environment; 2) the application of the approach to the SEL production environment yielded the characteristic software metric set {source lines, design effort, number of I/O parameters, fault correction effort per executable statement, code effort, number of versions}; and 3) the use of a characteristic set with conditional probabilities from historical data can assist in project management by forecasting the outcome of system parts. This work is intended to advance the understanding of the use of various metrics to characterize and predict aspects of software cost and quality.

5. Conclusions

The understanding of the technologies that contribute to quality in the software development process and the final product is fundamental to the advancement of the software field. This dissertation presents three studies that evaluate factors in key areas of software development, maintenance, and management: testing strategies, Cleanroom software development, and environmental metrics.

In each of the studies, a seven-step approach for quantitatively evaluating software technologies couples software methodology evaluation with software measurement. In the approach, goal/question frameworks of a technology's potential effect on software cost and quality are coupled with measurable attributes and appropriate quantitative analysis methods. The seven-step analysis methodology provides a paradigm for quantitatively assessing the effect of software technologies on software development and maintenance.

The goal structure, data analysis, and conclusions were presented for three studies: a blocked subject-project study comparing software testing strategies, a replicated project study characterizing the effect of using the Cleanroom software development approach, and a multi-project variation study to determine a characteristic set of software cost and quality metrics. The different studies were chosen to satisfy several criteria: scope of evaluation, representative domain sampling, quantitative analysis method, area of assessment, scope of technology, and potential benefit. The three studies are the following. 1) Soft-

ware Testing Strategies: A 74-subject study, including 32 professional programmers and 42 advanced university students, compared code reading, functional testing, and structural testing in a fractional factorial design. 2) Cleanroom Software Development: Fifteen three-person teams separately built a 1200-line message system to compare Cleanroom software development (in which software is developed completely off-line) with a more traditional approach. 3) Characteristic Software Metric Sets: In the NASA S.E.L. production environment, a study of 65 candidate product and process measures of 652 modules from six (51,000 - 112,000 line) projects yielded a characteristic set of software cost/quality metrics.

These empirical studies are intended to demonstrate an analysis methodology in a variety of problem domains and to advance the understanding of 1) the contribution of various software testing strategies to the software development process and to one another; 2) the relationship between introducing discipline into the development process and several aspects of product quality (requirement conformance, operational reliability, and modifiable source code); and 3) the use of software metrics to characterize software environments and to predict project outcome.

5.1. Overall Results from the Software Technology Evaluations

The major results from the software technology evaluations are the following. 1) With the professionals programmers, code reading detected more software faults and had a higher fault detection rate than did functional or struc-

tural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 2) With the advanced students, the three testing techniques were not different in the number of faults detected or in the fault detection rate, except that structural testing detected fewer faults than did the others in one study phase. 3) Code reading detected more interface faults and functional testing detected more control faults than did the other methods. 4) Most developers using the Cleanroom software development approach were able to build systems completely off-line. 5) The Cleanroom teams' products met system requirements more completely and succeeded on more operational test cases than did those developed with a traditional approach. 6) An approach described for calculating a characteristic metric set yielded the set for the NASA S.E.L. environment {source lines, design effort, number of input/output parameters, fault correction effort per executable statement, code effort, number of versions}.

5.2. Problem Areas

The use of the quantitative approach for evaluating software technologies identified several problem areas in data collection and analysis in software research and management, suggesting future research areas. 1) The process of formulating intuitive problems into precisely stated goals is a nontrivial task. The inherent difficulty in goal writing reflects the uncertainty of all aspects of quality in the software product and development process. 2) Numerous software metrics have been proposed to measure distinct attributes of software.

These metrics need to be validated to determine whether they actually capture what is intended. 3) The process of collecting accurate data is a continuing challenge. While there is increasing potential in automated collection schemes, the more common data collection forms are subject to incompleteness, inconsistency, and human error. 4) With the growing number of controlled studies done to determine which factors contribute to software quality, the selection of samples (e.g., programmers, programs, ...) to analyze is fundamental. In order for the results of these studies to apply to larger environments, representative samples of sufficient size must be selected. 5) These controlled studies are expensive to conduct. Both industry and academia must help support these efforts; e.g., academic researchers using subjects from industry. 6) There seems to be an interdependency among several factors that contribute to product and process quality. The use of several techniques together may be effective as a "critical mass", making the isolation of their individual effects difficult. 7) The methods of analysis must account for the high variation in individual performance. Without careful planning, this many-to-one differential among humans can taint experimental results. 8) Researchers have rarely been able to reproduce results across environments. In addition to the lack of consistent use of measures, every software development or modification environment seems to differ.

5.3. Overall Conclusions

The quantitative approach for evaluating software technologies has been applied in three analyses of factors contributing to software quality. The overall conclusions from this work are the following. 1) The approach described for quantitatively evaluating software technologies has been demonstrated and effective in a variety of problem domains. 2) The results from the testing strategy study suggest that code reading by stepwise abstraction (a nonexecution-based method) is at least as effective as on-line functional and structural testing in terms of number and cost of faults observed. 3) The results from the Cleanroom study demonstrate the feasibility of complete off-line development (as in Cleanroom) and suggest that such a development approach is superior to a more traditional approach. 4) The results from the software metric study suggest that a characteristic metric set can assist in aspects of project management, including the forecasting of effort for development, modification, and fault correction of modules based on historical data.

6. Appendices

6.1. Appendix A. Overview of Sampling and Statistical Test Application

In the range of software analyses in the four-part classification scheme presented earlier, there is a relationship between the effectiveness of statistical methods (attainable statistical significance) and the representativeness of the sampled observations to production-world situations. Because of this observers sometimes criticize the conclusions of an analysis or express doubt as to how well the results would extrapolate to environments different that the ones studied. This happens even when the analysis presented was sound and statistically significant. Emphasis needs to be placed on *two aspects of applying statistical* tools in an analysis, observation sampling and statistical test application. When an experiment is run, a certain sampling of data from some population is analyzed to achieve some result. After applying a statistical test to attributes of the members of the sample, a set of conclusions is derived.

The major considerations when choosing a sampling from a population are how well the sampling represents the whole population and how large the sampling should be. If a population is finite, the most representative sampling would be to select the whole population. There could then be no argument that the observations studied did not represent the whole population. Several interesting populations, such as programmers or software systems, are infinite so a reasonable finite sampling must be chosen. Techniques used to effectively

choose this sample are in statistical sampling theory [Cochran 53]. The stratification of the population is an important aspect of this process; that is, the identification of all the relevant aspects that differentiate among members of the population. This set of aspects is then distilled into a pseudo-basis [16] set, and then observations are chosen along the range of each basis set component. If statistical results are generated from a finite sampling of an infinite population, the issue of controversy is usually how well this sampling corresponds to the intended population. One component of the representativeness of this set is its size.

In determining the sample size, both the achievement of statistical significance in the experimental design and the economic constraints need to be considered. The cardinality of the basis set determines the number of factors whose effect must be accounted for in the experimental design. The effect of these factors is blocked out in the design, enabling the investigation to focus on distinguishing between the particular treatments being examined [Box, Hunter, & Hunter 78]. The decision to choose an experimental design is balanced between one capable of blocking out these factors and the need to keep the sample size economically feasible. The size of the sample also effects the probability of erroneous conclusions, referred to as Type I and Type II errors [Siegel 55, pp. 8-11]. [17]

---

[16] The prefix pseudo is used here since the basis set achieved is usually an approximation of a true basis set.

[17] Type I error is rejecting the experimental hypothesis when it is indeed true. The probability of Type I error is the significance level, usually called al-

When applying a statistical test to a set of data, any assumptions that the test requires must be verified. For example, assumptions regarding the distribution of values or their variance commonly occur in parametric statistics. Given that a set of data meets the required assumptions, the determination of the outcome of the test is just mathematics; criticizing this aspect of experimentation is unfounded. Note that different statistical tests have their own characteristics, such as in terms of the power or sensitivity of the test [Siegel 55, pp. 10-11]. [18] Given that the assumptions for two different tests are both met, one of the tests may be more appropriate to be chosen on these or other grounds.

6.2. Appendix B. Programs Used in the Testing Strategy Comparison

6.2.1. Appendix B.1. The Specifications for the Programs

Program 1

Given an input text of up to 80 characters consisting of words separated by blanks or new-line characters, the program formats it into a line-by-line form such that 1) each output line has a maximum of 30 characters, 2) a word in the input text is placed on a single output line, and 3) each output line is filled with as many words as possible.

---

pha. Type II error is not rejecting the experimental hypothesis when it is false. The probability of Type II error is usually called beta.

[18] The power of a test is one minus the probability of Type II error.

The input text is a stream of characters, where the characters are categorized as either break or nonbreak characters. A break character is a blank, a new-line character (&), or an end-of-text character (/). New-line characters have no special significance; they are treated as blanks by the program. The characters & and / should not appear in the output.

A word is defined as a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters and is reduced to a single blank character or start of a new line in the output.

When the program is invoked, the user types the input line, followed by a / (end-of-text) and a carriage return. The program then echos the text input and formats it on the terminal.

If the input text contains a word that is too long to fit on a single output line, an error message is typed and the program terminates. If the end-of-text character is missing, an error message is issued and the program awaits the input of properly terminated line of text.

Program 2

Given ordered pairs (x,y) of either positive or negative integers as input, the program plots them on a grid with a horizontal x-axis and a vertical y-axis which are appropriately labeled. A plotted point on the grid should appear as an asterisk (*).

The vertical and horizontal scaling is handled as follows. If the maximum absolute value of any y-value is less than or equal to twenty (20), the scale for

vertical spacing will be one line per integral unit (e.g., the point (3,6) should be plotted on the sixth line; two lines above the point (3,4)). Note that the origin (point (0,0)) would correspond to an asterisk at the the intersection of the axes (the x-axis is referred to as the 0th line). If the maximum absolute value of any x-value is less than or equal to thirty (30), the scale for horizontal spacing will be one space per integral unit (e.g., the point (4,5) should be plotted four spaces to the right of the y-axis; two spaces to the right of (2,5)). However, if the maximum absolute value of any y-value is greater than twenty (20), the scale for vertical spacing will be one line per every (max abs of yval)/20 rounded-up. (e.g., If the maximum absolute value of any y-value to be plotted is 66, the vertical line spacing will be a line for every four (4) integral units. In such a data set, points with y-values greater than or equal to eight and less than twelve will show up as asterisks in the second line, points with y-values greater than or equal to twelve and less than sixteen will show up as asterisks in the third line, etc. Continuing the example, the point (3,15) should be plotted on the third line; two lines above the point (3,5).) Horizontal scaling is handled analogously.

If two or more of the points to be plotted would show up as the same asterisk in the grid (like the points (9,13) and (9,15) in the above example), a number '2' (or whatever number is appropriate) should be printed instead of the asterisk. Points whose asterisks will lie on a axis or grid marker should show up in place of the marker.

Program 3

A list is defined to be an ordered collection of integer elements which may have elements annexed and deleted at either end, but not in the middle. The operations that need to be available are ADDFIRST, ADDLAST, DELETEFIRST, DELETELAST, FIRST, ISEMPTY, LISTLENGTH, REVERSE, and NEWLIST. Each operation is described in detail below. The lists are to contain up to a maximum of five (5) elements. If an element is added to the front of a "full" list (one containing five elements already), the element at the back of the list is to be discarded. Elements to be added to the back of a full list are discarded. Requests to delete elements from empty lists result in an empty list, and requests for the first element of an empty list results in zero (0) being returned. The detailed operation descriptions are as below:

ADDFIRST(LIST L, INTEGER I)
Returns the list L with I as its first element followed by all the elements of L. If L is "full" to begin with, L's last element is lost.

ADDLAST(LIST L, INTEGER I)
Returns the list with all of the elements of L followed by I. If L is full to begin with, L is returned (i.e., I is ignored).

DELETEFIRST(LIST L)
Returns the list containing all but the first element of L. If L is empty, then an empty list is returned.

DELETELAST(LIST L)
Returns the list containing all but the last element of L. If L is empty, then an empty list is returned.

FIRST(LIST L)
Returns the first element in L. If L is empty, then it returns zero (0).

ISEMPTY(LIST L)
Returns one (1) if L is empty, zero (0) otherwise.

LISTLENGTH(LIST L)
Returns the number of elements in L. An empty list has zero (0) elements.

NEWLIST(LIST L)
Returns an empty list.

REVERSE(LIST L)

Returns a list containing the elements of L in reverse order.

Program 4

(Note that a 'file' is the same thing as an IBM 'dataset'.)

The program maintains a database of bibliographic references. It first reads a master file of current references, then reads a file of reference updates, merges the two, and produces an updated master file and a cross reference table of keywords.

The first input file, the master, contains records of 74 characters with the following format:

column	comment
1 - 3	each reference has a unique reference key
4 - 14	author of publication
15 - 72	title of publication
73 - 74	year issued

The key should be a three (3) character unique identifier consisting of letters between A-Z. The next input file, the update file, contains records of 75 characters in length. The only difference from a master file record is that an update record has either an 'A' (capital A meaning add) or a 'R' (capital R meaning replace) in column 75. Both the master and update files are expected to be already sorted alphabetically by reference key when read into the program. Update records with action replace are substituted for the matching key record in

the master file. Records with action add are added to the master file at the appropriate location so that the file remains sorted on the key field. For example, a valid update record to be read would be (including a numbered line just for reference)

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
BITbaker        an introduction to program testing                    83A
```

The program should produce two pieces of output. It should first print the sorted list of records in the updated master file in the same format as the original master file. It should then print a keyword cross reference list. All words greater than three characters in a publication's title are keywords. These keywords are listed alphabetically followed by the key fields from the applicable updated master file entries. For example, if the updated master file contained two records,

```
ABCkermit       introduction to software testing                      82
DDXJones        the realities of software management                  81
```

then the keywords are introduction, testing, realities, software, and management. The cross reference list should look like

```
introduction
    ABC
management
    DDX
```

realities

DDX

software

ABC

DDX

testing

ABC

Some possible error conditions that could arise and the subsequent actions include the following. The master and update files should be checked for sequence, and if a record out of sequence is found, a message similar to 'key ABC out of sequence' should appear and the record should be discarded. If an update record indicates replace and the matching key can not be found, a message similar to 'update key ABC not found' should appear and the update record should be ignored. If an update record indicates add and a matching key is found, something like 'key ABC already in file' should appear and the record should be ignored. (End of specification.)

6.2.2. Appendix B.2. The Source Code for the Programs

Program 1

```
001: C NOTE THAT YOU DO NOT NEED TO VERIFY THE FUNCTION 'MATCH'
002: C   IT IS DESCRIBED THE FIRST TIME IT IS USED, AND ITS SOURCE CODE
003: C   IS INCLUDED AT THE END FOR COMPLETENESS.
004: C
005: C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS INCLUDE
       A LEADING
006: C   AND REQUIRED ' ' FOR CARRIAGE CONTROL
007:
008: C VARIABLE USED IN FIRST, BUT NEEDS TO BE INITIALIZED
009:       INTEGER MOREIN
```

```
010:
011: C STORAGE USED BY GCHAR
012:       INTEGER BCOUNT
013:       CHARACTER*1 GBUFER(80)
014:       CHARACTER*80 GBUF
015: C GBUFER AND GBUFSTR ARE EQUIVALENCED
016:
017: C STORAGE USED BY PCHAR
018:       INTEGER I
019:       CHARACTER*1 OUTLIN(31)
020: C OUTLIN AND OUTLINST ARE EQUIVALENCED
021:
022:       CHARACTER*1 GCHAR
023:
024: C CONSTANT USED THROUGHOUT THE PROGRAM
025:       CHARACTER*1 EOTEXT, BLANK, LINEFD
026:       INTEGER MAXPOS
027:
028:       COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
029:      X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF
030:
031:       DATA EOTEXT, BLANK, LINEFD, MAXPOS / '/', ' ', '&', 31 /
032:
033:
034:       CALL FIRST
035:       END
036:
037:
038:       SUBROUTINE FIRST
039:       INTEGER K, FILL, BUFPOS
040:       CHARACTER*1 CW
041:       CHARACTER*1 BUFFER(31)
042:
043:       INTEGER MOREIN, BCOUNT, I, MAXPOS
044:       CHARACTER*1 OUTLIN(31), GCHAR, EOTEXT, BLANK, LINEFD,
045:      X         GBUFER(80)
046:       CHARACTER*80 GBUF
047:
048:       COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
049:      X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF
050:
051:       BUFPOS = 0
052:       FILL = 0
053:       CW = ' '
054:
055:       MOREIN = 1
056:
057:       I = 1
058:       K = 1
059:       DOWHILE (K .LE. MAXPOS)
060:            OUTLIN(K) = ' '
061:            K = K + 1
062:       ENDDO
063:
064:       BCOUNT = 1
```

```
065:       K = 1
066:       DOWHILE (K .LE. 80)
067:            GBUFER(K) = 'Z'
068:            K = K + 1
069:       ENDDO
070:
071:       DOWHILE (MOREIN)
072:            CW = GCHAR()
073:            IF ((CW .EQ. BLANK) .OR. (CW .EQ. LINEFD) .OR.
074:      X                  (CW .EQ. EOTEXT)) THEN
075:                 IF (CW .EQ. EOTEXT)  THEN
076:                      MOREIN = 0
077:                 ENDIF
078:                 IF ((FILL+1+BUFPOS) .LE. MAXPOS)  THEN
079:                      CALL PCHAR(BLANK)
080:                      FILL = FILL + 1
081:                   ELSE
082:                      CALL PCHAR(LINEFD)
083:                      FILL =0
084:                 ENDIF
085:                 K = 1
086:                 DOWHILE (K .LE. BUFPOS)
087:                      CALL PCHAR(BUFFER(K))
088:                      K = K + 1
089:                 ENDDO
090:                 FILL = FILL + BUFPOS
091:                 BUFPOS = 0
092:              ELSE
093:                 IF (BUFPOS .EQ. MAXPOS)  THEN
094:                      WRITE(6,10)
095: 10                    FORMAT(' ','***WORD TO LONG***')
096:                      MOREIN = 1
097:                   ELSE
098:                      BUFPOS = BUFPOS + 1
099:                      BUFFER(BUFPOS) = CW
100:                 ENDIF
101:            ENDIF
102:       ENDDO
103:       CALL PCHAR(LINEFD)
104:       END
105:
106:
107:       CHARACTER*1 FUNCTION GCHAR()
108:       INTEGER MATCH
109:       CHARACTER*80 GBUFSTR
110:
111:       INTEGER MOREIN, BCOUNT, I, MAXPOS
112:       CHARACTER*1 OUTLIN(31), EOTEXT, BLANK, LINEFD,
113:      X         GBUFER(80)
114:       CHARACTER*80 GBUF
115:       COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
116:      X    EOTEXT, BLANK, LINEFD, GBUFER, GBUF
117:
118:       EQUIVALENCE (GBUFSTR,GBUFER)
119:
```

```
120:       IF (GBUFER(1) .EQ. 'Z')  THEN
121:            READ(5,20) GBUF
122: 20          FORMAT(A80)
123: C
124: C MATCH(CARRAY,C) RETURNS 1 IF CHARACTER C IS IN
       CHARACTER ARRAY
125: C   CARRAY, RETURNS 0 OTHERWISE. ARSIZE IS THE SIZE OF CARRAY.
126: C
127:            IF (MATCH(GBUF,EOTEXT) .EQ. 0)  THEN
128:                 WRITE(6,30)
129: 30               FORMAT(' ','***NO END OF TEXT MARK***')
130:                 GBUFER(2) = EOTEXT
131:               ELSE
132: C                GBUFER(1) = GBUF
133:                 GBUFSTR = GBUF
134:            ENDIF
135:       ENDIF
136:       GCHAR = GBUFER(BCOUNT)
137:       BCOUNT = BCOUNT + 1
138:       END
139:
140:
141:       SUBROUTINE PCHAR (C)
142:       CHARACTER*1 C
143:       CHARACTER*31 SOUT, OUTLINST
144:       INTEGER K
145:
146:       INTEGER MOREIN, BCOUNT, I, MAXPOS
147:       CHARACTER*1 OUTLIN(31), GCHAR, EOTEXT, BLANK, LINEFD,
148:      X        GBUFER(80)
149:       CHARACTER*80 GBUF
150:       COMMON /ALL/ MOREIN, BCOUNT, I, MAXPOS, OUTLIN,
151:      X   EOTEXT, BLANK, LINEFD, GBUFER, GBUF
152:
153:       EQUIVALENCE (OUTLINST,OUTLIN)
154:
155:       IF (C .EQ. LINEFD)  THEN
156:            SOUT = OUTLINST
157:            WRITE(6,40) SOUT
158: 40          FORMAT(' ',A31)
159:            K = 1
160:            DOWHILE (K .LE. MAXPOS)
161:                 OUTLIN(K) = ' '
162:                 K = K + 1
163:            ENDDO
164:            I = 1
165:          ELSE
166:            OUTLIN(I) = C
167:            I = I + 1
168:       ENDIF
169        END
```

Program 2

```
1: INT WIDTH = 30,
```

```
 2:    HEIGHT = 20,
 3:    GRIDWD = 61,
 4:    LARGENUM = 100000000
 5: STRING TICKS[61] =
 6:   '|-  -|-  -|-  -|-  -|-  -|-  -|-  -|-  -|-  -|-  -|-  -|-  -|'
 7:
 8:
 9: PROC SORT (INT ARRAY KEYBUF, INT ARRAY FREEBUF, INT N)
10:
11:    INT I, MAXP
12:    INT ARRAY SRTKEYB(100), SRTFREEB(100)
13:
14:    I := 0
15:    WHILE I < N DO
16:       SRTKEYB(I) := KEYBUF(I)
17:       SRTFREEB(I) := FREEBUF(I)
18:       I := I + 1
19:       END
20:
21:    I := N
22:    WHILE I > 0 DO
23:       MAXP := MAXELE(SRTKEYB,I)
24:       KEYBUF(N-I) := SRTKEYB(MAXP)
25:       FREEBUF(N-I) := SRTFREEB(MAXP)
26:       CALL REMOVE(SRTKEYB,MAXP,I)
27:       CALL REMOVE(SRTFREEB,MAXP,I)
28:       I := I - 1
29:       END
30:
31:
32:
33: INT FUNC MAXELE (INT ARRAY BUF, INT N)
34:
35:    INT I, MAXPTR, MAX
36:
37:    MAXPTR := -1
38:    MAX := -LARGENUM
39:    I := 0
40:    WHILE I < N DO
41:       IF BUF(I) > MAX
42:          THEN
43:          MAX := BUF(I)
44:          MAXPTR := I
45:          END
46:       I := I + 1
47:       END
48:    RETURN(MAXPTR)
49:
50:
51:
52: INT FUNC MINELE (INT ARRAY BUF, INT N)
53:
54:    INT I, MINPTR, MIN
55:
56:    MINPTR := -1
```

```
57:     MIN := LARGENUM
58:     I := 0
59:     WHILE I < N DO
60:        IF BUF(I) < MIN
61:           THEN
62:           MIN := BUF(I)
63:           MINPTR := I
64:           END
65:        I := I + 1
66:        END
67:     RETURN(MINPTR)
68:
69:
70:
71: PROC REMOVE (INT ARRAY BUF, INT PTR, INT N)
72:
73:     INT I
74:
75:     I := PTR
76:     WHILE I < N-1 DO
77:        BUF(I) := BUF(I+1)
78:        I := I + 1
79:        END
80:
81:
82:
83: INT FUNC ABS (INT VAL)
84:
85:     IF VAL < 0
86:        THEN
87:        RETURN(-VAL)
88:        ELSE
89:        RETURN(VAL)
90:        END
91:
92:
93:
94: INT FUNC SLASH (INT TOP, INT BOT)
95:
96:     INT RES
97:
98:     RES := TOP/BOT
99:     IF TOP <> RES*BOT .AND.
100:          (TOP > 0 .AND. BOT > 0 .OR. TOP < 0 .AND. BOT < 0)
101:       THEN RES := RES + 1
102:       END
103:    RETURN(RES)
104:
105: INT FUNC MOD (INT N, INT M)
106:
107:    INT VAL
108:
109:    VAL := N-N/M*M
110:    IF VAL < 0
111:       THEN
```

```
112:       VAL := VAL + M
113:       END
114:    RETURN (VAL)
115:
116:
117: PROC MAIN
118:
119:    CHAR ARRAY GRID(61)
120:    STRING STR[61]
121:    INT ARRAY XVAL(100), YVAL(100)
122:    INT I, J, NUMOBS, MAXY, MAXX, MINX, HORISP, VERTSP, VLINE
123:
124:    I := 0
125:    WHILE .NOT. EOI DO
126:       READ(XVAL(I),YVAL(I))
127:       I := I + 1
128:       END
129:    NUMOBS := I
130:
131:    CALL SORT(YVAL,XVAL,NUMOBS)
132:    MAXY := YVAL(0)
133:    VERTSP := SLASH(MAXY,HEIGHT)
134:
135:    MAXX := XVAL(MAXELE(XVAL,NUMOBS))
136:    MINX := XVAL(MINELE(XVAL,NUMOBS))
137:    IF ABS(MINX) > ABS(MAXX)
138:       THEN
139:       HORISP := SLASH(ABS(MINX),WIDTH)
140:       ELSE
141:       HORISP := SLASH(ABS(MAXX),WIDTH)
142:       END
143:
144:    STR := '                    X AXIS'
145:    WRITE(STR,SKIP)
146:    I := 0
147:    VLINE := HEIGHT
148:    WHILE VLINE > 0 DO
149:
150:       J := 0
151:       IF MOD(VLINE,5) = 0
152:          THEN
153:          UNPACK(TICKS,GRID)
154:          ELSE
155:          WHILE J < GRIDWD DO
156:             GRID(J)  = " "
157:             J := J + 1
158:             END
159:          END
160:
161:       VLINE := VLINE - 1
162:
163:       WHILE VLINE*VERTSP < YVAL(I) DO
164:          IF XVAL(I) >= 0
165:             THEN
166:             GRID(WIDTH - SLASH(XVAL(I),HORISP)) = "*"
```

```
167:              ELSE
168:              GRID(WIDTH - SLASH(-XVAL(I),HORISP)) := "*"
169:              END
170:           I := I + 1
171:           END
172:
173:        GRID(WIDTH) := "|"
174:        PACK(GRID,STR)
175:        WRITE(STR,SKIP)
176:        END
177:
178:     STR :=
      '|---|---|---|---|---|---|---|---|---|---|---|---|---|'
179:     UNPACK(STR,GRID)
180:     WHILE 0 <= YVAL(I) .AND. I <= NUMOBS DO
181:        IF XVAL(I) >= 0
182:           THEN
183:           GRID(WIDTH + SLASH(XVAL(I),HORISP)) := "*"
184:           ELSE
185:           GRID(WIDTH - SLASH(-XVAL(I),HORISP)) := "*"
186:           END
187:        I := I + 1
188:        END
189:
190:     PACK(GRID,STR)
191:     WRITE(STR,SKIP)
192:     STR := '                                    Y AXIS'
193:     WRITE(STR,SKIP)
194:
195: START MAIN
```

Program 3

```
001: C NOTE THAT YOU DO NOT NEED TO VERIFY THE FUNCTIONS
       DRIVER, GETARG,
002: C   CHAREQ, CODE, AND PRINT. THEIR SOURCE CODE IS
       DESCRIBED AND
003: C   INCLUDED AT THE END FOR COMPLETENESS.
004: C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS
       INCLUDE A LEADING
005: C   AND REQUIRED ' ' FOR CARRIAGE CONTROL
006: C
007:       INTEGER POOL(7), LSTEND
008:       INTEGER LISTSZ
009: C
010:       COMMON /ALL/ LISTSZ
011: C
012: C
013:       LISTSZ = 5
014:       CALL DRIVER (POOL, LSTEND)
015:       STOP
016:       END
017: C
018: C
019:       FUNCTION ADFRST (POOL, LSTEND, I)
```

```
020:        INTEGER ADFRST
021:        INTEGER POOL(7), LSTEND, I
022:        INTEGER LISTSZ
023:        COMMON /ALL/ LISTSZ
024: C
025:        INTEGER A
026: C
027:        IF (LSTEND .GT. LISTSZ) THEN
028:            LSTEND = LISTSZ - 1
029:        ENDIF
030:        LSTEND = LSTEND + 1
031:        A = LSTEND
032:        DOWHILE (A .GE. 1)
033:            POOL(A+1) = POOL(A)
034:            A = A - 1
035:        ENDDO
036: C
037:        POOL(1) = I
038:        ADFRST = LSTEND
039:        RETURN
040:        END
041: C
042: C
043:        FUNCTION ADLAST (POOL, LSTEND, I)
044:        INTEGER ADLAST
045:        INTEGER POOL(7), LSTEND, I
046:        INTEGER LISTSZ
047:        COMMON /ALL/ LISTSZ
048: C
049:        IF (LSTEND .LE. LISTSZ) THEN
050:            LSTEND = LSTEND + 1
051:            POOL(LSTEND) = I
052:        ENDIF
053:        ADLAST = LSTEND
054:        RETURN
055:        END
056: C
057: C
058:        FUNCTION DELFST (POOL, LSTEND)
059:        INTEGER DELFST
060:        INTEGER POOL(7), LSTEND
061:        INTEGER LISTSZ
062:        COMMON /ALL/ LISTSZ
063: C
064:        INTEGER A
065:        IF (LSTEND .GT. 1) THEN
066:            A = 1
067:            LSTEND = LSTEND - 1
068:            DOWHILE (A .LE. LSTEND)
069:                POOL(A) = POOL(A+1)
070:                A = A + 1
071:            ENDDO
072:        ENDIF
073:        DELFST = LSTEND
074:        RETURN
```

```
075:        END
076: C
077: C
078:        FUNCTION DELLST (LSTEND)
079:        INTEGER DELLST
080:        INTEGER LSTEND
081: C
082:        IF (LSTEND .GE. 1) THEN
083:            LSTEND = LSTEND - 1
084:        ENDIF
085:        DELLST = LSTEND
086:        RETURN
087:        END
088: C
089: C
090:        FUNCTION FIRST (POOL, LSTEND)
091:        INTEGER FIRST
092:        INTEGER POOL(7), LSTEND
093: C
094:        IF (LSTEND .LE. 1) THEN
095:            FIRST = 0
096:          ELSE
097:            FIRST = POOL(1)
098:        ENDIF
099:        RETURN
100:        END
101: C
102: C
103:        FUNCTION EMPTY (LSTEND)
104:        INTEGER EMPTY
105:        INTEGER LSTEND
106: C
107:        IF (LSTEND .LE. 1) THEN
108:            EMPTY = 1
109:          ELSE
110:            EMPTY = 0
111:        ENDIF
112:        RETURN
113:        END
114: C
115: C
116:        FUNCTION LSTLEN  (LSTEND)
117:        INTEGER LSTLEN
118:        INTEGER LSTEND
119: C
120:        LSTLEN = LSTEND - 1
121:        RETURN
122:        END
123: C
124: C
125:        FUNCTION NEWLST (LSTEND)
126:        INTEGER NEWLST
127:        INTEGER LSTEND
128: C
129:        NEWLST = 0
```

```
130:        RETURN
131:        END
132: C
133: C
134:        SUBROUTINE REVERS (POOL, LSTEND)
135:        INTEGER POOL(7), LSTEND
136: C
137:        INTEGER I, N
138: C
139:        N = LSTEND
140:        I = 1
141:        DOWHILE (I .LE. N)
142:             POOL(I) = POOL(N)
143:             N = N - 1
144:             I = I + 1
145:        ENDDO
146:        RETURN
147:        END
```

Program 4

```
001: C NOTE THAT YOU DO NOT NEED TO VERIFY THE ROUTINES
       DRIVER, STREQ, WORDEQ,
002: C  NXTSTR, ARRCPY, CHARPT, BEFORE, CHAREQ, AND WRDBEF.
       THEIR SOURCE
003: C  CODE IS DESCRIBED AND INCLUDED AT THE END FOR
       COMPLETENESS.
004: C NOTE THAT FORMAT STATEMENTS FOR WRITE STATEMENTS
       INCLUDE A LEADING
005: C  AND REQUIRED ' ' FOR CARRIAGE CONTROL
006: C THE SFORT LANGUAGE CONSTRUCT '.IF (EXPRESSION)' BEGINS
       A BLOCKED
007: C  IF-THEN[-ELSE] STATEMENT, AND IT IS EQUIVALENT TO
       THE F77
008: C  'IF (EXPRESSION) THEN'.
009: C
010:      CALL DRIVER
011:      STOP
012:      END
013: C
014: C
015:      SUBROUTINE MAINSB
016: C
017:      LOGICAL*1 U$KEY(3),U$AUTH(11),U$TITL(58),U$YEAR(2),U$ACTN(1)
018:      LOGICAL*1 M$KEY(3),M$AUTH(11),M$TITL(58),M$YEAR(2)
019:      LOGICAL*1 ZZZ(3), LASTUK(3), LASTMK(3)
020:      LOGICAL*1 STREQ, CHAREQ, BEFORE, CHARPT
021:      INTEGER I
022: C
023:      LOGICAL*1 WORD(500,12), REFKEY(1000,3)
024:      INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
025:      COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
026: C
027:      WRITE(6,290)
028: 290  FORMAT(' ','   UPDATED LIST OF MASTER ENTRIES')
```

```
029:      DO 300 I = 1, 3
030:         LASTMK(I) = CHARPT(' ')
031:         LASTUK(I) = CHARPT(' ')
032:         ZZZ(I) = CHARPT('Z')
033: 300   CONTINUE
034: C
035:      NUMWDS = 0
036:      NUMREF = 0
037:      CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
038:      CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
039: C
040:      DOWHILE ((.NOT.(STREQ(M$KEY,ZZZ,3))) .OR.
041:     X         (.NOT.(STREQ(U$KEY,ZZZ,3))) )
042:         .IF (STREQ(U$KEY,M$KEY,3))
043:            .IF (.NOT.(CHAREQ(U$ACTN(1),'R')))
044:               WRITE(6,100) U$KEY
045: 100              FORMAT(' ','KEY ',3A1,' IS ALREADY IN FILE')
046:            ENDIF
047:            CALL OUTPUT(U$KEY,U$AUTH,U$TITL,U$YEAR)
048:            CALL DICTUP(U$KEY,U$TITL,58)
049:            CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
050:            CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
051:         ENDIF
052: C
053:         .IF (BEFORE(M$KEY,3,U$KEY,3))
054:            CALL OUTPUT(M$KEY,M$AUTH,M$TITL,M$YEAR)
055:            CALL DICTUP(M$KEY,M$TITL,58)
056:            CALL GETNM(M$KEY,M$AUTH,M$TITL,M$YEAR,LASTMK)
057:         ENDIF
058: C
059:         .IF (BEFORE(U$KEY,3,M$KEY,3))
060:            .IF (CHAREQ(U$ACTN(1),'R'))
061:               WRITE(6,110) U$KEY
062: 110              FORMAT(' ','UPDATE KEY ',3A1,' NOT FOUND')
063:            ENDIF
064:            CALL OUTPUT(U$KEY,U$AUTH,U$TITL,U$YEAR)
065:            CALL DICTUP(U$KEY,U$TITL,58)
066:            CALL GETNUP(U$KEY,U$AUTH,U$TITL,U$YEAR,U$ACTN,LASTUK)
067:         ENDIF
068:      ENDDO
069: C
070:      CALL SRTWDS
071:      CALL PRTWDS
072:      RETURN
073:      END
074: C
075: C
076:      SUBROUTINE GETNM(KEY,AUTH,TITL,YEAR,LASTMK)
077:      LOGICAL*1 KEY(3),AUTH(11),TITL(58),YEAR(2),LASTMK(3)
078: C
079:      LOGICAL*1 SEQ, INLINE(80)
080:      LOGICAL*1 BEFORE, CHARPT, CHAREQ
081:      LOGICAL*1 GO$M, GO$U
082:      COMMON /DRIV/ GO$M, GO$U
083: C
```

```
084:      SEQ = 1
085:      DOWHILE (SEQ)
086:         .IF (GO$M)
087: C
088: C READ FROM THE MASTER FILE
089: C
090:            READ(10,200,END=299) INLINE
091:          ELSE
092: C
093: C SEE REMARK ABOUT THE CHARACTER '%' LATER IN THE ROUTINE.
094: C
095:            INLINE(1) = CHARPT('%')
096:         ENDIF
097: 200      FORMAT(80A1)
098:         DO 210 I = 1, 3
099:            KEY(I) = INLINE(I)
100: 210      CONTINUE
101:         DO 220 I = 1, 11
102:            AUTH(I) = INLINE(3+I)
103: 220      CONTINUE
104:         DO 230 I = 1, 58
105:            TITL(I) = INLINE(14+I)
106: 230      CONTINUE
107:         DO 240 I = 1, 2
108:            YEAR(I) = INLINE(72+I)
109: 240      CONTINUE
110: C
111: C A METHOD OF SPECIFYING END-OF-FILE IN A FILE IS TO PUT
       THE CHARACTER '%'
112: C   AS THE FIRST CHARACTER ON A LINE.  THE DRIVER USES THIS
       FOR MULTIPLE
113: C   SETS OF INPUT CASES.
114: C
115:         .IF ((.NOT.(CHAREQ(KEY(1),'%'))) .AND.
116:     X      (BEFORE(KEY,3,LASTMK,3)) )
117:            WRITE(6,250) KEY
118: 250         FORMAT(' ','KEY ',3A1,' OUT OF SEQUENCE')
119:          ELSE
120:            CALL ARRCPY(KEY,LASTMK,3)
121:            SEQ = 0
122:         ENDIF
123:         .IF (CHAREQ(KEY(1),'%'))
124:            SEQ = 0
125:            DO 270 I = 1, 3
126:               KEY(I) = CHARPT('Z')
127: 270         CONTINUE
128:         ENDIF
129:      ENDDO
130:      RETURN
131: 299   CONTINUE
132:      GO$M = 0
133:      DO 260 I = 1, 3
134:         KEY(I) = CHARPT('Z')
135: 260   CONTINUE
136:      RETURN
```

```
137:      END
138: C
139: C
140:      SUBROUTINE GETNUP(KEY,AUTH,TITL,YEAR,ACTN,LASTUK)
141:      LOGICAL*1 KEY(3),AUTH(11),TITL(58),YEAR(2),ACTN(1),LASTUK(3)
142: C
143:      LOGICAL*1 SEQ, INLINE(80)
144:      LOGICAL*1 BEFORE, CHARPT, CHAREQ
145:      LOGICAL*1 GO$M, GO$U
146:      COMMON /DRIV/ GO$M, GO$U
147: C
148:      SEQ = 1
149:      DOWHILE (SEQ)
150:         .IF (GO$U)
151: C
152: C READ FROM THE UPDATES FILE
153: C
154:            READ(11,200,END=299) INLINE
155:          ELSE
156: C
157: C SEE REMARK ABOUT THE CHARACTER '%' LATER IN THE ROUTINE.
158: C
159:            INLINE(1) = CHARPT('%')
160:         ENDIF
161: 200      FORMAT(80A1)
162:         DO 210 I = 1, 3
163:            KEY(I) = INLINE(I)
164: 210      CONTINUE
165:         DO 220 I = 1, 11
166:            AUTH(I) = INLINE(3+I)
167: 220      CONTINUE
168:         DO 230 I = 1, 58
169:            TITL(I) = INLINE(14+I)
170: 230      CONTINUE
171:         DO 240 I = 1, 2
172:            YEAR(I) = INLINE(72+I)
173: 240      CONTINUE
174:         ACTN(1) = INLINE(75)
175: C
176: C A METHOD OF SPECIFYING END-OF-FILE IN A FILE IS TO PUT
        THE CHARACTER '%'
177: C   AS THE FIRST CHARACTER ON A LINE.  THE DRIVER USES THIS
        FOR MULTIPLE
178: C   SETS OF INPUT CASES.
179: C
180:         .IF ((.NOT.(CHAREQ(KEY(1),'%'))) .AND.
181:      X     (BEFORE(KEY,3,LASTUK,3)) )
182:            WRITE(6,250) KEY
183: 250         FORMAT(' ','KEY ',3A1,' OUT OF SEQUENCE')
184:          ELSE
185:            CALL ARRCPY(KEY,LASTUK,3)
186:            SEQ = 0
187:         ENDIF
188:         .IF (CHAREQ(KEY(1),'%'))
189:            SEQ = 0
```

```
190:              DO 270 I = 1, 3
191:                 KEY(I) = CHARPT('Z')
192: 270           CONTINUE
193:            ENDIF
194:         ENDDO
195:         RETURN
196: 299     CONTINUE
197:         GO$U = 0
198:         DO 260 I = 1, 3
199:            KEY(I) = CHARPT('Z')
200: 260     CONTINUE
201:         RETURN
202:         END
203: C
204: C
205:         SUBROUTINE OUTPUT(KEY,AUTH,TITL,YEAR)
206:         LOGICAL*1 KEY(3), AUTH(11), TITL(58), YEAR(2)
207: C
208:         WRITE(6,200) KEY, AUTH, TITL, YEAR
209: 200     FORMAT(' ',3A1,11A1,58A1,2A1)
210:         RETURN
211:         END
212: C
213: C
214:         SUBROUTINE PRTWDS
215: C
216:         LOGICAL*1 WORD(500,12), REFKEY(1000,3)
217:         INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
218:         COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
219: C
220: C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A LINKED
       LIST.
221: C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
222: C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS
       AS A
223: C   KEYWORD WORD(I,J),J=1,12
224: C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY
       THAT HAS
225: C   AS A KEYWORD WORD(I,J),J=1,12
226: C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
227: C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER
       KEYS FOR
228: C   THE PARTICULAR KEYWORD
229: C
230:         INTEGER I, J
231:         LOGICAL*1 FLAG
232: C
233:         WRITE(6,200)
234: 200     FORMAT(' ','   KEYWORD REFERENCE LIST')
235:         DO 210 I = 1, NUMWDS
236:            FLAG = 1
237:            WRITE(6,220) (WORD(I,J),J=1,12)
238: 220        FORMAT(' ',12A1)
239:            LAST = PTR(I)
240:            DOWHILE (FLAG)
```

```
241:            WRITE(6,230) (REFKEY(LAST,J),J=1,3)
242: 230         FORMAT(' ','    ',3A1)
243:            LAST = NEXT(LAST)
244:            .IF (LAST .EQ. -1)
245:               FLAG = 0
246:            ENDIF
247:          ENDDO
248: 210    CONTINUE
249:       RETURN
250:       END
251: C
252: C
253:       SUBROUTINE DICTUP(KEY,STR,STRLEN)
254:       LOGICAL*1 KEY(3), STR(120)
255:       INTEGER STRLEN
256: C
257:       LOGICAL*1 WDLEFT, FLAG, OKLEN, NEXTWD(120), WORDEQ
258:       INTEGER LPTR, NXTSTR, LEN, LAB, I, K
259: C
260:       LOGICAL*1 WORD(500,12), REFKEY(1000,3)
261:       INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
262:       COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
263: C
264: C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A
       LINKED LIST.
265: C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
266: C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS
       AS A
267: C   KEYWORD WORD(I,J),J=1,12
268: C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY
       THAT HAS
269: C   AS A KEYWORD WORD(I,J),J=1,12
270: C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
271: C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER
       KEYS FOR
272: C   THE PARTICULAR KEYWORD
273: C
274:       WDLEFT = 1
275:       LPTR = 1
276: C
277:       DOWHILE (WDLEFT)
278:          FLAG = 1
279:          OKLEN = 1
280:          LEN = NXTSTR(STR,STRLEN,LPTR,NEXTWD,120)
281:          .IF (LEN .EQ. 0)
282:             WDLEFT = 0
283:          ENDIF
284: C
285:          .IF (LEN .LE. 2)
286:             OKLEN = 0
287:          ENDIF
288: C
289:          .IF (OKLEN)
290:             I = 1
291:             DOWHILE ((I .LE. NUMWDS).AND. FLAG )
```

```
292:                .IF (WORDEQ(NEXTWD,I))
293:                   LAB = I
294:                   FLAG = 0
295:                ENDIF
296:                I = I + 1
297:             ENDDO
298:             .IF (FLAG)
299:                NUMWDS = NUMWDS + 1
300:                NUMREF = NUMREF + 1
301:                DO 300 K = 1, 12
302:                   WORD(NUMWDS,K) = NEXTWD(K)
303: 300            CONTINUE
304:                PTR(NUMWDS) = NUMREF
305:                DO 310 K = 1, 3
306:                  REFKEY(NUMREF,K) = KEY(K)
307: 310            CONTINUE
308:                NEXT(NUMREF) = -1
309:             ELSE
310:                NUMREF = NUMREF + 1
311:                DO 320 K = 1, 3
312:                   REFKEY(NUMREF,K) = KEY(K)
313: 320            CONTINUE
314:                NEXT(NUMREF) = PTR(LAB)
315:                PTR(LAB) = NUMREF
316:             ENDIF
317:          ENDIF
318:       ENDDO
319: C
320:       RETURN
321:       END
322: C
323: C
324:       SUBROUTINE SRTWDS
325: C
326:       LOGICAL*1 WORD(500,12), REFKEY(1000,3)
327:       INTEGER NUMWDS, NUMREF, PTR(500), NEXT(1000)
328:       COMMON /WORDS/ WORD, REFKEY, NUMWDS, NUMREF, PTR, NEXT
329: C
330: C THE ABOVE GROUP OF DATA STRUCTURES SIMULATES A
       LINKED LIST.
331: C WORD(I,J) IS A KEYWORD -- J RANGING FROM 1 TO 12
332: C REFKEY(PTR(I),K),K=1,3 IS THE FIRST 3 LETTER KEY THAT HAS
       AS A
333: C   KEYWORD WORD(I,J),J=1,12
334: C REFKEY(NEXT(PTR(I)),K),K=1,3 IS THE SECOND 3 LETTER KEY
       THAT HAS
335: C   AS A KEYWORD WORD(I,J),J=1,12
336: C REFKEY(NEXT(NEXT(PTR(I))),K),K=1,3 IS THE THIRD ... ETC.
337: C NEXT(J) IS EQUAL TO -1 WHEN THERE ARE NO MORE 3 LETTER
       KEYS FOR
338: C   THE PARTICULAR KEYWORD
339: C
340:       INTEGER I, J, K, LAB, LOWERB, UPPERB
341:       LOGICAL*1 WRDBEF, NEXTWD(12)
342: C
```

```
343:      UPPERB = NUMWDS - 1
344:      DO 400 I = 1, UPPERB
345:         LOWERB = I + 1
346:         DO 410 J = LOWERB, NUMWDS
347:            IF (WRDBEF(J,I))
348:               DO 300 K = 1, 12
349:                  NEXTWD(K) = WORD(I,K)
350: 300           CONTINUE
351:               LAB = PTR(I)
352:               DO 310 K = 1, 12
353:                  WORD(I,K) = WORD(J,K)
354: 310           CONTINUE
355:               PTR(I) = PTR(J)
356:               DO 320 K = 1, 12
357:                  WORD(J,K) = NEXTWD(K)
358: 320           CONTINUE
359:               PTR(J) = LAB
360:            ENDIF
361: 410     CONTINUE
362: 400  CONTINUE
363: C
364:      RETURN
365:      END
```

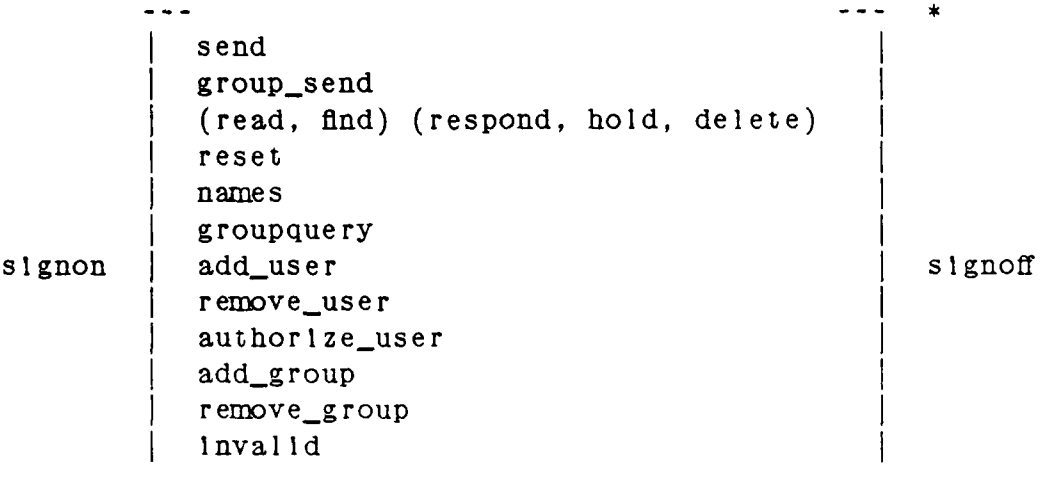
6.3. Appendix C. Operational Testing Procedure Applied in the Cleanroom Study

This section describes the operational testing process applied to the projects in the Cleanroom empirical investigation. After consulting the references [Thayer, Lipow & Nelson 78, Duran & Ntafos 81, Dyer 82a, Dyer 82a, Dyer 82b]., the following procedure was adopted to meet the particular circumstances.

6.3.1. Test Data Selection

The first step in the test data generation process is to define the operational profile of the system. An initial attempt to define the operational inputs to the message system and their serialization requirements resulted in the regular expression in Figure 37.

Figure 37. Regular expression of logical inputs to the system in a single user session.



This then led to a transition diagram of functional paths through the system. There were distinct transition arcs in the diagram to correspond with distinct functional states of the system. The system states were described as either system processing or operating states. A distinction in the processing of data that is transparent to the user is a system "processing state" (e.g., whether or not a target's queue is empty). A distinction in the processing of data that a user was directly responsible for is a system "operating state" (e.g., giving an incorrect password). The arcs leaving a given node were each assigned a frequency such that the total of all outgoing arcs from a given node was one. This frequency assignment was accomplished by polling eleven well-seasoned users of the University of Maryland Vax 11/780 mailing system. Now that each path through the system had a (subjective) probability, the schedule of presentation

and the test sample size needed to be considered.

The sequence of releases and their associated functionality for one of the teams appears in Figure 38.

Figure 38. Schedule of Deliveries for a Sample Team.

Groupings of Capabilities

B_00 signon,signoff
B_0 add_user,remove_user,authorize_user
B_1 names,invalid_cmds
B_2 send_msg,read_msg,find_msg,hold_msg, del_msg, respond,reset_queue
B_3 group_send,group_add,group_remove, group_query

Release	Function Group				
	B_00	B_0	B_1	B_2	B_3
1	XX				
2	XX	XX			
3	XX	XX		XX	
4	XX	XX	XX	XX	
5	XX	XX	XX	XX	XX

(Note that since each team could chose its own delivery schedule, the test generation process needed to be reevaluated for each team.) The graph from Figure 37 has been cut and reconstructed according to the groupings of capabilities given in Figure 38. Be aware that the probability for any given path through the system is preserved in such a process. The newly created arcs for the

groups are labeled with the probabilities that a given system input will invoke any function in a particular group (see bottom of Figure 39). Notice that for any sample size of less than 200, the expected value of the number of test cases invoking a "privileged" function, group B_0, would be less than one. Since the relative input frequency for the "privileged" group of commands is disproportionate to their importance (you would not be able to build and maintain a network of subscribers without them), a separate schedule for testing them is created. Figure 39 shows the two schedules for testing.

Figure 39. Two Testing Schedules for a Sample Team.

Release	Function Group				
Schedule I	B_00*	B_0	B_1	B_2	B_3
1	XX				
2	XX	XX			
Schedule II					
3	XX	XX		XX	
4	XX	XX	XX	XX	
5	XX	XX	XX	XX	XX
Operational frequency		.005	.129	.686	.180 1.000

*Note that the functions in B_00 are implicitly tested in all test cases.

Since Schedule I is a special case, first consider Schedule II with the function group probabilities at the bottom of the columns. In order to accomplish the concentration of test cases on the newly released functions, each functional group is assigned a relative weighting that it should have in the test subset selections for each of the releases. The weights in each of the columns should sum to one.

Release	Function Group			
	B_0	B_1	B_2	B_3
3	1/3	0	1/2	0
4	1/3	2/3	1/3	0
5	1/3	1/3	1/6	1
	.005	.129	.686	.180

The weights in a given column are then multiplied by the total associated frequency for that functional group at the bottom of the column.

Release	Function Group			
	B_0	B_1	B_2	B_3
3	1/3 .001	0 0.0	1/2 .343	0 0.0
4	1/3 .002	2/3 .086	1/3 .229	0 0.0
5	1/3 .002	1/3 .043	1/6 .114	1 .180
	.005	.129	.686	.180

The entries (not the weights) in the above table are the probabilities that an input will be selected to test a function in a given group on a given release. Sum-

ming the rows horizontally reflects the total distribution across the releases.

Release	Function Group								
	B_0		B_1		B_2		B_3		
3	1/3	.001	0	0.0	1/2	.343	0	0.0	.344
4	1/3	.002	2/3	.086	1/3	.229	0	0.0	.317
5	1/3	.002	1/3	.043	1/6	.114	1	.180	.339
		.005		.129		.686		.180	1.000

The above process represents the partitioning on the input frequencies for the various functional groupings by release.

Hopefully at some time we will be able to specify the size of our test sample from the reliability goals of the project. For the purposes of testing these projects, our experience has led us to choose a test sample size of 100 cases per project. If ten of these cases (arbitrarily) will be used in the testing of Schedule I, ninety will remain for Schedule II. Multiplying ninety by the frequencies in the right hand column of the above table for Schedule II led to the sample sizes of 31, 29 and 30 test cases for releases 3, 4 and 5, respectively. The above process has been undertaken to test the expanding system capabilities, while concentrating on newly released functions and maintaining the composite input distribution. Figure 40 summarizes the results from this stratification process. In testing release 1, only the signon and signoff functions (group B_00) were available and hence only one test case is needed. The remaining nine test cases are applied to release 2 to test the group of "privileged" functions B_0. The arcs

B_00, ..., B_3 are reassigned the frequencies given in Figure 40 when test data are being generated for the appropriate release.

Figure 40. Arc Frequency Assignment as a Result of Stratification.

Release	Arc Frequency Assignment for Function Group					#Test Cases
	B_00*	B_0	B_1	B_2	B_3	
1	1.0	0.0	0.0	0.0	0.0	1
2	0.0	1.0	0.0	0.0	0.0	9
3	0.0	.003	0.0	.997	0.0	31
4	0.0	.006	.271	.723	0.0	29
5	0.0	.006	.127	.336	.531	30

						100

*Recall that the functions in B_00 are implicitly tested in all test cases.

After this moderately complex procedure, test data can finally be created. With respect to the revised arc frequency assignments above, a set of test data of the appropriate size is randomly generated for each release.

6.3.2. Testing Process and Failure Observation

The actual testing process consists of three phases for each test case: system "state" setup (recall the system processing and operating "states" described earlier), executing the actual test, and verifying the result of the test. Since our concern in the reliability analysis is with failure-free execution intervals, the cpu-time for just the second phase, the actual test case execution, is included in our calculation.

The projects developed were tested interactively, with each given test case having one of four possible outcomes. If the system performed to expectations on the test case, the outcome was a 'success.' If the system's performance did not meet expectations, the outcome was a 'failure' and was rated according to severity: 1 - product inoperable, 2 - major function in the product inoperable, 3 - some part of a major function inoperable, or 4 - cosmetic type failure. If the outcome was a 'failure' but the same failure was observed on an earlier test case in this release, the outcome was termed a 'duplicate failure.' Finally, if the test case was not able to be executed because we were unable to create the proper system "state" (on account of failures in this release), the outcome of the test case was 'deferred.' Test cases with outcomes of 'failure,' 'duplicate failure,' or 'deferred' were included in the test set of the next release.

6.3.3. Failure Counting

Several software reliability models are based on a product's history of failure-free execution intervals [Jelinski & Moranda 73, Dyer & Mills 82, Goel 82]. In order to calculate these intervals, a consistent interpretation of what constitutes a failure must be determined. A method of "sorting" or masking failures by associated product release, product function or by failure severity has been recognized [Dyer 81]. This technique enables calculation of reliability estimates for certain functions within a system, including only those failures worse than a certain severity, etc. In addition to these options, a more fundamental set of questions needs to be considered. Such as, whether or not dupli-

cate failures should be counted, or whether the execution time for regression (previously failed) tests should be included. Several of these failure counting issues are summarized in Figure 41.

Figure 41. Failure Counting Issues.

Always include cpu time in failure-free interval for (unless masked)
- successful non-regression tests
- first occurrence of distinct failures

Never include cpu time for
- deferred test cases

Options:

A. Include cpu time from regression tests:
   1. just from successful?
   2. just from failed?

B. Duplicate failures:
   1. include duplicate failures observed in the same release?
   2. include duplicate failures observed in later releases?

C. Execution interval that terminated with end of testing (assuming did not end with a failure):
   1. discard?
   2. include as failure-free execution interval -- treat end of testing as a failure?
   3. include as failure-free execution interval of twice the length -- treat end of testing as a failure twice as far off?

D. Masking:
   1. by testing schedule?
   2. by product release?
   3. by product function?
   4. by failure severity?

---

7. References

[Adams 84]

E. N. Adams, Optimizing Preventive Service of Software Products, *IBM Journal of Research and Development* **28**, 1, pp. 2-14, Jan. 84.

[Albin & Ferreol 82]

J.-L. Albin and R. Ferreol, Collecte et analyse de mesures de logiciel (Collection and Analysis of Software Data), *Technique et Science Informatiques* **1**, 4, pp. 297-313, 1982. (Rairo ISSN 0752-4072)

[Bailey & Basili 81]

J. W. Bailey and V. R. Basili, A Meta-Model for Software Development Resource Expenditures, *Proc. Fifth Int. Conf. Software Engr.*, San Diego, CA, pp. 107-116, 1981.

[Bailey 84]

J. W. Bailey, Teaching Ada: A Comparison of Two Approaches, Dept. Com. Sci., Univ. Maryland, College Park, MD, working paper, 1984.

[Baker 72a]

F. T. Baker, System Quality Through Structured Programming, *AFIPS Proc. 1972 Fall Joint Computer Conf.* **41**, pp. 339-343, 1972.

[Baker 72b]

F. T. Baker, Chief Programmer Team Management of Production Programming, *IBM Systems J.* **11**, 1, pp. 131-149, 1972.

[Baker 81]

F. T. Baker, Chief Programmer Teams, pp. 249-254 in *Tutorial on Structured Programming: Integrated Practices*, ed. V. R. Basili and F. T. Baker, IEEE, 1981.

[Basili et al. 85]

V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. L. Ramsey, and S. Chang, A Quantitative Characterization and Evaluation of a Software Development in Ada, (to appear *IEEE Computer*, September 1985)

[Basili & Turner 76]

V. R. Basili and A. J. Turner, *SIMPL-T: A Structured Programming Language*, Paladin House Publishers, Geneva, IL, 1976.

[Basili et al. 77]
V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, Jr., W. F. Truszkowski, and D. L. Weiss, The Software Engineering Laboratory, Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-77-001, May 1977.

[Basili & Zelkowitz 78]
V. R. Basili and M. V. Zelkowitz, Analyzing Medium-Scale Software Developments, *Proc. Third Int. Conf. Software Engr.*, Atlanta, GA, pp. 116-123, May 1978.

[Basili 80]
Victor R. Basili, *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Computer Society, New York, 1980.

[Basili & Freburger 81]
V. R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *Journal of Systems and Software* **2**, pp. 47-57, 1981.

[Basili & Weiss 81]
V. R. Basili and D. M. Weiss, Evaluation of a Software Requirements Document By Analysis of Change Data, *Proc. Fifth Int. Conf. Software Engr.*, San Diego, CA, pp. 314-323, March 9-12, 1981.

[Basili & Reiter 81]
V. R. Basili and R. W. Reiter, A Controlled Experiment Quantitatively Comparing Software Development Approaches, *IEEE Trans. Software Engr.* **SE-7**, May 1981.

[Basili & Doerflinger 83]
V. R. Basili and C. Doerflinger, Monitoring Software Development Through Dynamic Variables, *Proc. COMPSAC*, Chicago, IL, 1983.

[Basili, Selby & Phillips 83]
V. R. Basili, R. W. Selby, Jr., and T. Y. Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, *IEEE Trans. Software Engr.* **SE-9**, 6, pp. 652-663, Nov. 1983.

[Basili & Hutchens 83]
V. R. Basili and D. H. Hutchens, An Empirical Study of a Syntactic Metric Family, *Trans. Software Engr.* **SE-9**, 6, pp. 664-672, Nov. 1983.

[Basili & Perricone 84]
V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM* **27**, 1, pp. 42-52, Jan. 1984.

[Basili & Selby 84]
V. R. Basili and R. W. Selby, Jr., Data Collection and Analysis in Software Research and Management, *Proceedings of the American Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, August 13-16, 1984.

[Basili & Ramsey 84]
V. R. Basili and J. R. Ramsey, Structural Coverage of Functional Testing, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1442, Sept. 1984.

[Basili & Weiss 84]
V. R. Basili and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data*, *Trans. Software Engr.* **SE-10**, 6, pp. 728-738, Nov. 1984.

[Behrens 83]
C. A. Behrens, Measuring the Productivity of Computer Systems Development Activities with Function Points, *IEEE Trans. Software Engr.* **SE-9**, 6, pp. 648-651, Nov. 1983.

[Boehm 81]
B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Boehm et al. 84]
B. W. Boehm, T. E. Gray, and T. Seewaldt, Prototyping Versus Specifying: A Multiproject Experiment, *IEEE Trans. Software Engr.* **SE-10**, 3, pp. 290-303, May 1984.

[Bowen 84]
J. Bowen, Estimation of Residual Faults and Testing Effectiveness, *Seventh Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 24-27, 1984.

[Box, Hunter, & Hunter 78]
G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*, John Wiley & Sons, New York, 1978.

[Brooks 80]
R. E. Brooks, Studying Programmer Behavior: The Problem of Proper Methodology, *Communications of the ACM* **23**, 4, pp. 207-213, 1980.

[Brooks 81]
W. D. Brooks, Software Technology Payoff: Some Statistical Evidence, *J. Systems and Software* **2**, pp. 3-9, 1981.

[Buck 81]
F. O. Buck, Indicators of Quality Inspections, IBM Systems Products Division, Kingston, NY, Tech. Rep. 21.802, Sept. 1981.

[Callilau & Rubin 79]
R. Callilau and F. Rubin, ACM Forum: On a Controlled Experiment in Program Testing, *Communications of the ACM* **22**, pp. 687-8, Dec. 1979.

[Card et al. 82]
D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili, The Software Engineering Laboratory, Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD Rep. SEL-81-104, Feb. 1982.

[Chen 78]
E. T. Chen, Program Complexity and Programmer Productivity, *IEEE Trans. Software Engr.*, pp. 187-194, May 1978.

[Church 84]
V. Church, Benchmark Statistics for the VAX 11/780 and the IBM 4341, Computer Sciences Corporation, Silver Spring, MD, Internal Memo, 1984.

[Cochran & Cox 50]
W. G. Cochran and G. M. Cox, *Experimental Designs*, John Wiley & Sons, New York, 1950.

[Cochran 53]
W. G. Cochran, *Sampling Techniques*, John Wiley & Sons, Inc., 1953.

[Currit 83]
P. A. Currit, Cleanroom Certification Model, *Proc. Eight Ann. Software Engr. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.

[Curtis et al. 79]
B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics, *IEEE Trans. Software Engr.*, pp. 96-104, March 1979.

[Curtis, Sheppard & Milliman 79]
B. Curtis, S. B. Sheppard, and P. M. Milliman, Third Time Charm: Stronger Replication of the Ability of Software Complexity Metrics to Predict Programmer Performance, *Proc. Fourth Int. Conf. Software Engr.*, pp. 356-360, Sept. 1979.

[Curtis 83]
B. Curtis, Cognitive Science of Programming, *Sixth Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 19-22, 1983.

[Decker & Taylor 82]
W. J Decker and W. A. Taylor, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-78-102, May 1982.

[Duran & Ntafos 81]
J. W. Duran and S. Ntafos, A Report on Random Testing*, *Proc. Fifth Int. Conf. Software Engr.*, San Diego, CA, pp. 179-183, March 9-12, 1981.

[Dyer 81]
M. Dyer, Cleanroom Project Management Data, IBM-FSD Internal Memo to H. D. Mills, October 16, 1981.

[Dyer 82a]
M. Dyer, An Approach to Statistical Testing for Cleanroom Software Development. IBM-FSD Tech. Rep. 86.0002, 1982.

[Dyer & Mills 82]
M. Dyer and H. D. Mills, Developing Electronic Systems with Certifiable Reliability, *Proc. NATO Conf.*, Summer, 1982.

[Dyer 82a]
M. Dyer, Major System Mode 11 (MSM11) Testing, IBM-FSD Internal Memo to H. D. Mills, May 18, 1982.

[Dyer 82b]

M. Dyer, Top-Down Random Testing, IBM-FSD Internal Memo to H. D. Mills, June 21, 1982.

[Dyer 82c]

M. Dyer, Cleanroom Software Development Method, IBM Federal Systems Division, Bethesda, MD, October 14, 1982.

[Dyer 83]

M. Dyer, Software Validation in the Cleanroom Development Method, IBM-FSD Tech. Rep. 86.0003, August 19, 1983.

[Elshoff 84]

J. L. Elshoff, Characteristic Program Complexity Metrics, *Proc. Seventh Int. Conf. Software Engr.*, Orlando, FL. pp. 288-293, 1984.

[Endres 75]

A. Endres, An Analysis of Errors and their Causes in Systems Programs, *IEEE Trans. Software Engr.*, pp. 140-149, June 1975.

[Fagan 76]

M. E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, *IBM Sys. J.* **15**, 3, pp. 182-211, 1976.

[Ferrentino & Mills 77]

A. B. Ferrentino and H. D. Mills, State Machines and Their Semantics in Software Engineering, *Proc. IEEE COMPSAC*, 1977.

[Feuer & Fowlkes 79]

A. R. Feuer and E. B. Fowlkes, Some Results from an Empirical Study of Computer Software, *Proc. Fourth Int. Conf. Software Engr.*, pp. 351-355, 1979.

[Foster 80]

K. A. Foster, Error Sensitive Test Cases, *IEEE Trans. Software Engr.* **SE-6**, 3, pp. 258-264, 1980.

[Gaffney & Heller 80]

J. E. Gaffney and G. L. Heller, Macro Variable Software Models for Application to Improved Software Development Management, *Proc. Workshop on Quantitative Software Models for Reliability, Complexity and Cost*, IEEE Comput. Society, 1980.

[Gannon & Horning 75]
J. D. Gannon and J. J. Horning, The Impact of Language Design on the Production of Reliable Software, *Trans. Software Engr.* **SE-1**, pp. 179-191, 1975.

[Gannon 77]
J. D. Gannon, An Experimental Evaluation of Data Type Conventions, *Communications of the ACM* **20**, 8, pp. 584-595, 1977.

[Gannon et al. 83]
J. D. Gannon, E. E. Katz, and V. R. Basili, Characterizing Ada Programs: Packages, *The Measurement of Computer Software Performance*, Los Alamos National Laboratory, Aug. 1983.

[Gloss-Soler 79]
S. A. Gloss-Soler, The DACS Glossary: A Bibliography of Software Engineering Terms, Data & Analysis Center for Software, Griffiss Air Force Base, NY 13441, Rep. GLOS-1, Oct. 1979.

[Goel 82]
A. L. Goel, Software Reliability and Estimation Techniques, Rome Air Development Center, NY, Rep. RADC-TR-82-263, October 1982.

[Goel 83]
A. L. Goel, A Guidebook for Software Reliability Assessment, Dept. Industrial Engr. and Operations Research, Syracuse Univ., New York, Tech. Rep. 83-11, April 1983.

[Goodenough & Gerhart 75]
J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Trans. Software Engr.*, pp. 156-173, June 1975.

[Gould & Drongowski 74]
J. D. Gould and P. Drongowski, An Exploratory Study of Computer Program Debugging, *Human Factors* **16**, 3, pp. 258-277, 1974.

[Gould 75]
J. D. Gould, Some Psychological Evidence on How People Debug Computer Programs, *International Journal of Man-Machine Studies* **7**, pp. 151-182, 1975.

[Halstead 77]
M. H. Halstead, *Elements of Software Science*, North Holland, New York, 1977.

[Hamer & Frewin 82]
P. G. Hamer and G. D. Frewin, M. H. Halstead's Software Science -- A Critical Examination, *Proc. Sixth Int. Conf. Software Engr.*, Tokyo, Japan, pp. 197-206, Sept 13-16, 1982.

[Hetzel 76]
W. C. Hetzel, An Expermental Analysis of Program Verification Methods, Ph.D. Thesis, Univ. of North Carolina, Chapel Hill, 1976.

[Hoare 69]
C. A. R. Hoare, An Axiomatic Basis for Computer Programming, *Communications of the ACM* **12**, 10, pp. 576-583, Oct. 1969.

[Howden 76]
W. E. Howden, Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Software Engr.* **SE-2**, 3, Sept. 1976.

[Howden 78]
W. E. Howden, Algebraic Program Testing, *Acta Informatica* **10**, 1978.

[Howden 80]
W. E. Howden, Functional Program Testing, *IEEE Trans. Software Engr.* **SE-6**, pp. 162-169, Mar. 1980.

[Howden 81]
W. E. Howden, A Survey of Dynamic Analysis Methods, pp. 209-231 in *Tutorial: Software Testing & Validation Techniques, 2nd Ed.*, ed. E. Miller and W. E. Howden, 1981.

[Hutchens & Basili 83]
D. H. Hutchens and V. R. Basili, System Structure Analysis: Clustering With Data Bindings, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1310, August 1983.

[Hwang 81]
S-S. V. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection*, Dept. Com. Sci., Univ. of Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[IEEE 83]
IEEE. IEEE Standard Glossary of Software Engineering Terminology, Rep. IEEE-STD-729-1983, IEEE, 342 E. 47th St. New York, 1983.

[Jelinski & Moranda 73]
Z. Jelinski and P. B. Moranda, Applications of a Probability-Based Model to a Code Reading Experiment, *Proc. IEEE Symposium on Computer Software Reliability*, New York, pp. 78-81, IEEE, 1973.

[Jensen & Wirth 74]
K. Jensen and N. Wirth, *PASCAL User Manual and Report, 2nd Ed.*, Springer-Verlag, New York, 1974.

[Johnson, Draper & Soloway 83]
W. L. Johnson, S. Draper, and E. Soloway, An Effective Bug Classification Scheme Must Take the Programmer into Account, *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.

[Kelly 82]
J. P. J. Kelly, Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach, UCLA Ph.D. Thesis, 1982.

[Knight 84]
J. Knight, A Large Scale Experiment in N-Version Programming, *Proc. of the Ninth Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1984.

[Linger, Mills & Witt 79]
R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

[McCabe 76]
T. J. McCabe, A Complexity Measure, *IEEE Trans. Software Engr.* **SE-2**, 4, pp. 308-320, Dec. 1976.

[McMullin & Gannon 80]
P. R. McMullin and J. D. Gannon, Evaluating a Data Abstraction Testing System Based on Formal Specifications, Dept. Com. Sci., Univ. of Maryland, College Park, Tech. Rep. TR-993, Dec. 1980.

[Miara et al. 83]
R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, Program Indentation and Comprehensibility, *Communications of the ACM* **26**, 11, pp. 861-867, Nov. 1983.

[Mills 72a]
H. D. Mills, Mathematical Foundations for Structural Programming, IBM Report FSL 72-6021, 1972.

[Mills 72b]
H. D. Mills, Chief Programmer Teams: Principles and Procedures, IBM Corp., Gaithersburg, MD, Rep. FSC 71-6012, 1972.

[Mills 75]
H. D. Mills, How to Write Correct Programs and Know It, *Int. Conf. on Reliable Software*, Los Angeles, pp. 363-370, 1975.

[Moher & Schneider 82]
T. Moher and G. M. Schneider, Methodology and Experimental Research in Software Engineering, *International Journal of Man-Machine Studies* **16**, 1, pp. 65-87, 1982.

[Musa 75]
J. D. Musa, A Theory of Software Reliability and Its Application, *IEEE Trans. Software Engr.* **SE-1**, 3, pp. 312-327, 1975.

[Myers 76]
G. J. Myers, *Software Reliability: Principles & Practices*, John Wiley & Sons, New York, 1976.

[Myers 78]
G. J. Myers, A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections, *Communications of the ACM*, pp. 760-768, Sept. 1978.

[Myers 79]
G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

[Naur 69]
P. Naur, Programming by Action Clusters, *BIT* **9**, 3, pp. 250-258, 1969.

[Ostrand & Weyuker 83]
T. J. Ostrand and E. J. Weyuker, Collecting and Categorizing Software Error Data in an Industrial Environment, Dept. Com. Sci., Courant Inst. Math. Sci., New York Univ., NY, Tech. Rep. 47, August 1982 (Revised May 1983).

[Panzl 81]

D. J. Panzl, Experience with Automatic Program Testing, *Proc. NBS Trends and Applications*, Nat. Bureau Stds., Gaithersburg, MD, pp. 25-28, May, 28 1981.

[Parnas 72a]

D. L. Parnas, Some Conclusions from an Experiment in Software Engineering Techniques, *AFIPS Proc. 1972 Fall Joint Computer Conf.* **41**, pp. 325-329, 1972.

[Parnas 72b]

D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM* **15**, 12, pp. 1053-1058, 1972.

[Parnas 72c]

D. L. Parnas, A Technique for Module Specification With Examples, *Communications of the ACM* **15**, May 1972.

[Ramsey 84]

J. Ramsey, Structural Coverage of Functional Testing, *Seventh Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 24-27, 1984.

[SEL 82]

Annotated Bibliography of Software Engineering Laboratory (SEL) Literature, Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD Rep. SEL-82-006, Nov. 1982.

[Selby 83]

R. W. Selby, Jr., An Empirical Study Comparing Software Testing Techniques, *Sixth Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 19-22, 1983.

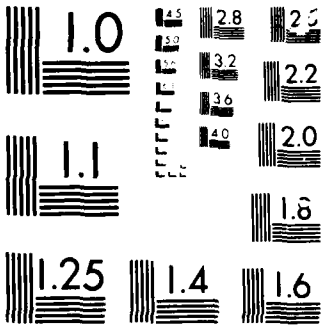
[Selby 84]

R. W. Selby, Jr., Evaluating Software Testing Strategies, *Proc. of the Ninth Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1984.

[Selby, Basili & Baker 85]

R. W. Selby, Jr., V. R. Basili, and F. T. Baker, CLEANROOM Software Development: An Empirical Evaluation, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1415, February 1985. (submitted to the *IEEE Trans. Software Engr.*)

AD-A168 738 EVALUATIONS OF SOFTWARE TECHNOLOGIES: TESTING CLEANROOM 3/3
AND METRICS(U) MARYLAND UNIV COLLEGE PARK DEPT OF
COMPUTER SCIENCE R W SELBY MAY 85 TR-1500
UNCLASSIFIED AFOSR-TR-86-0279 F49620-80-C-0001 F/G 9/2 NL



MICROCOPY CHART

[Shankar 82]
K. S. Shankar, A Functional Approach to Module Verification, *IEEE Trans. Software Engr.* **SE-8**, 2, March 1982.

[Shell 81]
B. A. Shell, The Psychological Study of Programming, *Computing Surveys* **13**, pp. 101-120, March 1981.

[Shen, Conte & Dunsmore 83]
V. Y. Shen, S. D. Conte, and H. E. Dunsmore, Software Science Revisited: A Critical Evaluation of the Theory and Its Empirical Support, *Trans. Software Engr.* **SE-9**, 2, pp. 155-165, March 1983.

[Shneiderman et al. 77]
B. Shneiderman, R. E. Mayer, D. McKay, and P. Heller, Experimental Investigations of the Utility of Detailed Flowcharts in Programming, *Communications of the ACM* **20**, 6, pp. 373-381, 1977.

[Siegel 55]
S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, New York, 1955.

[Soloway et al. 82]
E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, What Do Novices Know About Programming?, in *Directions in Human-Computer Interactions*, ed. A. Badre and B. Shneiderman, Ablex, Inc., 1982.

[Soloway 83]
E. Soloway, You Can Observe a Lot by Just Watching How Designers Design, *Proc. Eight Ann. Software Engr. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.

[Soloway & Ehrlich 84]
E. Soloway and K. Ehrlich, Empirical Studies of Programming Knowledge, *Trans. Software Engr.* **SE-10**, 5, pp. 595-609, Sept. 1984.

[Stuckl 77]
L. G. Stuckl, New Directions in Automated Tools for Improving Software Quality, in *Current Trends in Programming Methodology*, ed. R. T. Yeh, Prentice Hall, Englewood Cliffs, NJ, 1977.

[Thayer, Lipow & Nelson 78]
R. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability*, North-Holland, Amsterdam, 1978.

[Valdes & Goel 83]
P. M. Valdes and A. L. Goel, An Error-Specific Approach to Testing, *Proc. Eight Ann. Software Engr. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.

[Vessey & Weber 83]
I. Vessey and R. Weber, Some Factors Affecting Program Repair Maintenance: An Empirical Study, *Communications of the ACM* **26**, 2, pp. 128-134, Feb. 1983.

[Vosburgh et al. 84]
J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, Productivity Factors and Programming Environments, *Proc. Seventh Int. Conf. Software Engr.*, Orlando, FL, pp. 143-152, 1984.

[Walston & Felix 77]
C. E. Walston and C. P. Felix, A Method of Programming Measurement and Estimation, *IBM Systems J.* **16**, 1, pp. 54-73, 1977.

[Weiss & Basili 85]
D. M. Weiss and V. R. Basili, Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, *IEEE Trans. Software Engr.* **SE-11**, 2, pp. 157-168, February 1985.

[Weissman 74]
L. Weissman, Psychological Complexity of Computer Programs: An Experimental Methodology, *SIGPLAN Notices* **9**, 6, pp. 25 - 36, June 1974.

[Woodfield, Dunsmore & Shen 81]
S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, The Effect of Modularization and Comments on Program Comprehension, Dept. Com. Sci., Arizona St. Univ., Tempe, AZ, working paper, 1981.

[Zolnowski & Simmons 81]
J. C. Zolnowski and D. B. Simmons, Taking the Measure of Program Complexity, *Proc. National Computer Conference*, pp. 329-336, 1981.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ~~TR-1500~~ AFOSR-TR- 86-0279	2. GOVT ACCESSION NO. ADA168738 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) EVALUATIONS OF SOFTWARE TECHNOLOGIES: Testing, CLEANROOM, and Metrics	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard W. Selby, Jr.	8. CONTRACT OR GRANT NUMBER(s) ~~AFOSR~~-F 49620-80-C-001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR Bolling AFB Washington, D. C. 20332	12. REPORT DATE May 1985
	13. NUMBER OF PAGES 181
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The evaluation of software technologies suffers because of the lack of quantitative assessment of their effect on software development and modification. A seven-step approach for quantitatively evaluating software technologies couples software methodology evaluation with software measurement. The approach is applied in-depth in the following three areas. 1) Software Testing Strategies: A 74-subject study, including 32 professional programmers and 42 advanced university students, compared code reading, functional testing, and structural testing in a fractional factorial design. 2) Cleanroom Software Development: Fifteen three-

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

person teams separately built a 1200-line message system to compare Cleanroom software development (in which software is developed completely off-line) with a more traditional approach. 3) Characteristic Software Metric Sets: In the NASA S.E.L. production environment, a study of 65 candidate product and process measures of 652 modules from six (51,000 - 112,000 line) projects yielded a characteristic set of software cost/quality metrics.

The major results are the following. 1) The approach described for quantitatively evaluating software technologies has been demonstrated and effective in a variety of problem domains. 2) With the professionals, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 3) With the students, the three techniques were not different in the number of faults detected or in the fault detection rate, except that structural testing detected fewer faults than did the others in one study phase. 4) Code reading detected more interface faults and functional testing detected more control faults than did the other methods. 5) Most developers using the Cleanroom software development approach were able to build systems completely off-line. 6) The Cleanroom teams' products met system requirements more completely and succeeded on more operational test cases than did those developed with a traditional approach. 7) An approach described for calculating a characteristic metric set yielded the set for the NASA S.E.L. environment (source lines, design effort, number of input/output parameters, fault correction effort per executable statement, code effort, number of versions).

END

DTIC

7-86